Intel 80386 程序员参考手册

wizardforcel

Published with GitBook



目錄

Intel 80386 程序员参考手册

个人爱好与需要，只翻译了操作系统部分的重要章节，如有翻译的不恰当的地方，由于本人才疏学浅，请多见谅。

浙江大学 冯金生

Email: fenghappy2000@163.com

2005-12-11

第一章 80386介绍

80386是一款32位处理器，为多任务操作系统作了优化，为需要很高性能的应用程序而设计。32位寄存器和数据通道支持32位地址和数据原型。处理器可以寻址4G物理存储器和64T（2**64 字节）虚拟存储器。片上储存期管理部件包括地址转换寄存器，高级多任务固件，保护机制，以及页虚拟存储器。专用的调试寄存器即使在ROM程序中也能够提供数据和代码中断点。

1.1 该手册的组织结构

该书从5个方面阐述了80386的体系结构：

```
第I部分        — 应用程序
第II部分       — 系统编程
第III部分       — 兼容性
第IV部分       — 指令集
附录
```

上面的分类一方面取决于体系结构本身，一方面取决于使用这本书的不同方式。入下表所示，后面的2部分旨在作为参考手册来帮组那些致力于在80386上开发软件的程序员。前面的3部分则是理论说明，它们说明体系结构的用途，解释术语和概念，描述那些或与专属用途相关，或与专属体系结构相关的指令。

```
解释说明

第I部分        — 应用程序
第II部分       — 系统编程
第III部分       — 兼容性

参考

第IV部分       — 指令集
附录
```

前面的3部分陈述了80386 CPU的执行模式和保护特性。应用特性和系统特性的区别由80386的保护机制决定。保护机制的主要用意在于是操作系统免于应用的干扰；因此，处理器使得一些寄存器和和指令不能被应用程序访问。第I部分讨论的是可以被应用访问的特性；第II部分讨论的特性只能由授权的系统软件或在非保护模式下访问。

80386的处理模式同样控制着可访问的特性。80386有3种处理模式：

1．保护模式。 2．实地址模式。 3．虚拟8086模式。

保护模式是80386处理器的正常的32位环境。在这种模式下，所有的指令和特性均可使用。

上电后处理器即进入实地址模式（经常简称为“实模式”）。实地址模式下，80386看起来像是一个更快速的，增加了一些新指令的8086。大多数80386软件在实地址模式下只是进行一些初始化操作。

虚拟8086模式（也常被称作“V86模式”）是一种动态模式，在这种意思上说，处理器可以频繁，快速的在V86模式和保护模式之间切换。CPU从保护模式进入V86模式去执行8086程序，然后再离开V86模式，进入保护模式继续执行原来的80386程序。

在保护模式下可使用的任何特性在V86模式下对于所有程序来说是一样的。这些特性构成了第I部分。在保护模式下系统软件可以使用的其他一些特性组成了第II部分。第III部分阐述了实地址模式和V86模式，也包括如何执行一个32位和16位的混合程序。

所有模式下可使用 第I部分－应用程序 只能在保护模式下使用 第II部分－系统编程 兼容模式 第III部分－兼容性

1.1.1 第I部分－应用编程

这部分呈现的是通常被应用程序员使用的特性。

第2章－基本编程模型：介绍了存储器的组织结构模型。定义数据类型。列举应用程序使用的寄存器集合。介绍堆栈。说明字符串操作。定义指令的组成。解释了地址计算。介绍应用程序可能用到的阿中断和异常。

第3章－应用指令集：纵览应用程序经常使用的指令。将指令按照功能分组；例如，字符串指令被分在一组，控制－传输指令被分在另一组。解释指令的概念。关于单个指令的详细介绍被放到了第IV部分，指令集参考。

1.1.2 第II部分－系统编程

这部分阐述的特性通常被下面的人使用：写操作系统，设备驱动，调试器以及为80386保护模式下应用程序提供支持的其他软件的人。

第4章－系统架构：纵览系统程序员使用的各项特性。介绍在第I部分没有提及的寄存器和数据结构。介绍面向系统的指令以及它们支持的寄存器和数据结构上下文。指出在在哪些章节可以找到寄存器，数据结构以及指令的详细信息。

第5章－内存管理：阐述了支持虚拟存储器的数据结构，寄存器以及指令的详细结构，以及段和页的概念。解释了设计者怎样选择存储器的组织模型，从完全线性（“平坦模式”）到使用页和段。

第6章－保护：展开存储器管理特性，包括它应用与断和页时的保护。说明了特权等级，堆栈切换，指针检查以及用户和超级用户模式的实现。多任务的保护被推迟到后面的章节。

第7章－多任务：阐述了硬件如何通过上下文切换和任务间保护来支持多任务。

第8章－输入/输出：揭示I/O特性，包括I/O指令，I/O的保护，以及I/O许可映射。

第9章－异常和中断：解释了80386中断的基本运行机制。中断和异常怎样和保护相联系。讨论了所有可能的异常，列举了触发条件和需要处理的信息，以及异常的恢复。

第10章－初始化：定义了处理器在复位或上电后的状态。说明了为实地址模式和保护模式设置寄存器，标志位以及数据结构。包含一个初始化示例。

第11章－协处理和多处理：解释了支持数字协处理器和共享存储器的多CPU的指令。

第12章－调试：告诉我们怎样使用调试寄存器。

1.1.3 第III部分－兼容性

本书的其他章节基本上把处理器看作是32位机器，为了简单而省略了16操作部件。确实，80386是32位机器，但是它的设计完全支持16位的操作数和寻址。这部分通过对在32位程序中支持16位程序和16位操作数的说明完整的描述了80386的结构特性。所有的3种模式均可执行16位程序：保护模式可以直接执行80286保护模式下的16位程序，实地址模式执行8086程序和80286实地址模式程序，虚拟8086模式在多任务环境下和其他80386保护模式程序一起执行8086程序。另外，32位和16位模块可以在保护模式下与32位和16位操作混合使用。

第13章－执行80286保护模式代码：在保护模式下，80386完全可以执行80286保护模式代码，因为80286是80386的子集。

第14章－80386实地址模式：说明80386的实地址模式。在这种模式下，80386看起来更像是一个快速的实地址模式下的80286，或增加了新指令的快速8086。

第15章－虚拟8086模式：80386能够快速的在保护模式和虚拟8086模式之间切换。这使其有能力在32位“本地模式”程序之间执行8086的多道程序。

第16章－混合16位和32位编码：即使在同一个程序或任务内，80386也能够混合16位和32位编码。而且，任何给定的模块可以同时使用16位和32位操作数和地址。

1.1.4 第IV部分－指令集

第I，II，III部分给出了指令和体系结构特定细节相关的总体描述，这部分用字母序列出了指令集，为汇编程序员，编写调试器，编译器，操作系统的人等提供了具体细节。指令介绍包括操作的逻辑描述，对标志位的影响，设置标志位的影响，操作数或地址长度属性的影响，处理器模式的影响以及可能产生的异常。

1.1.5 附录

附录给出了编码表和其他细节，它们被组织成了可以被汇编程序员和系统程序员快速索引的形式。

1.2 其他文献

下面的书籍包含有关80386的其他资料。

- Introduction to the 80386, order number 231252
- 80386 Hardware Reference Manual, order number 231732
- 80386 System Software Writer's Guide, order number 231499
- 80386 High Performance 32-bit Microprocessor with Integrated Memory Management (Data Sheet), order number 231630

1.3 符号转换

本手册在描述数据结构，指令助记符，十六进制数以及上标和下标时用了特殊的符号。下标用{}括起来，例如10{2} = 10 以2为基。上标用()加上前面的^来表示，例如，10^(3) = 10的3次幂。复习这些符号有助于以后的阅读。

1.3.1 数据结构格式

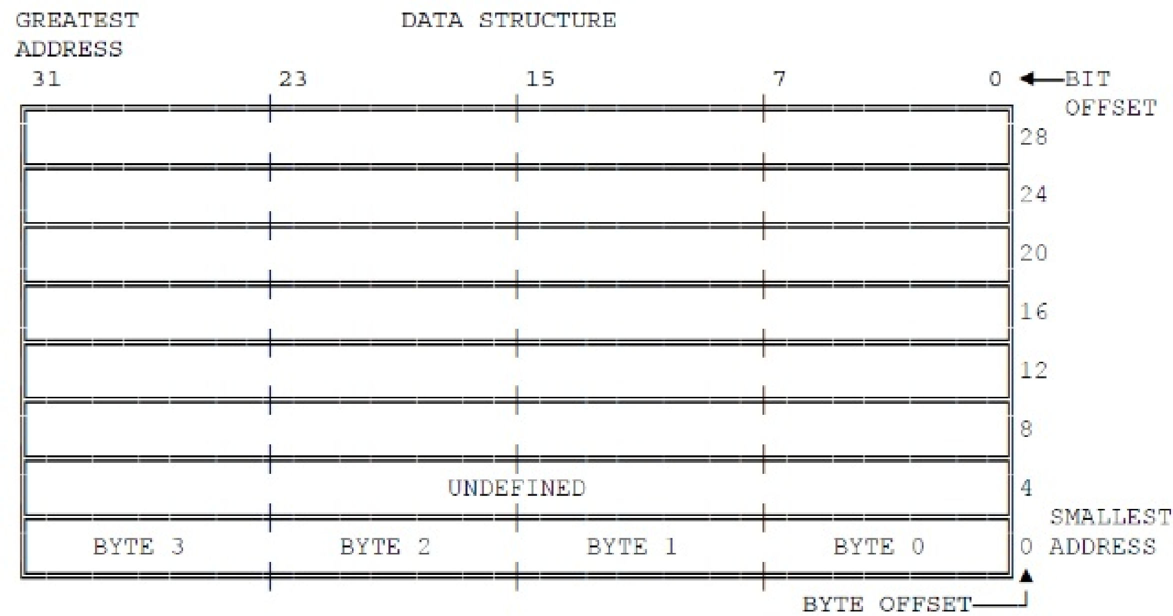
在内存数据结构的示例图中，低位地址出现在图示的右边；地址从右至左，从下往上递增。位从右向左依次编号。图1－1举例说明了这种惯例。

1.3.2 未定义位和软件兼容性

在许多寄存器和存储器的布局图中，一些位被标记为未定义。当位标记为未定义（如图1－1所示）时，将来的软件将这些位按未定义来处理对于软件兼容性非常重要。在处理未定义位时软件应该遵循下列规则：

- 在测试含有未定义位的寄存器时，不要依赖这些位的状态。在测试前要屏蔽掉这些未定义位。
- 在将寄存器的值拷贝到另一个寄存器时，不要依赖这些位。
- 不要依赖于保留在已写入未定义位的信息。
- 装载寄存器时，要始终把未定义位按0载入，或以之前存在寄存器中的值重新载入。

注意： 依赖于寄存器中的未定义位将导致软件依赖于80386在处理这些位时的未指定的处理方式。如果将来的处理器使用了未定义位，那么依赖于这些位的软件有不兼容的风险。任何软件都要避免依赖于未定义的80386寄存器位。



1.3.3 指令操作数

当用符号表示指令时，你正在使用的是80386汇编指令集。在这个集合中，指令遵循下面的格式：

```
标号：前缀 助记符 参数1，参数2，参数3
```

这里：

- 标号是指令的标识符，后面跟冒号。
- 前缀是一条指令前缀的可选保留名字。
- 助记符暗示指令执行的操作，操作码中的一个保留字。
- 操作数参数1，参数2，参数3为可选项。可以有0－3个参数，取决于指令码。当含有参数时，它们或者是立即数，或者是数据项的标识符。操作数标识符可以是寄存器的保留字，或者在其他程序中声明（在例子中可能没有这种形式），指向数据项。当修改数据的指令中含有两个操作数时，右边的是源操作数，左边的是目的操作数。

例子：

```
LOADREG: MOV EAX, SUBTOTAL
```

本例中，LOADREG是标号，MOV是操作码指令助记符，EAX是目的操作数，SUBTOTAL是源操作数。

1.3.4 十六进制数

十六进制数字后面加上H的字符串表示以16为基的数字。十六进制数字从下面的集合中选取（0，1，2，3，4，5，6，7，8，9, A, B, C, D, E, F）。某些情况下，特别是在程序语法例子中，一个前导零会被加在A-F前面。例如，0FH等于十进制的15。

1.3.5 上标和下标

本手册使用特殊的符号来表示上标和下标。下标用{}括起来，例如10{2} = 10 以2为基。上标用()加上前面的^来表示，例如，10^(3) = 10的3次幂。

编者注：本手册在适当的地方用实际形式来表示上标和下标。

第二章 编程基本模型

这章描述了处理器在保护模式下对汇编程序员可见的应用程序编程环境。向程序员介绍了直接影响应用程序设计和实现的80386特性。其他章节讨论和系统编程或与其他8086处理器家族兼容性相关的一些特性。

基本编程模型包括以下几个方面：

- 存储器组织和段
- 数据类型
- 寄存器
- 指令格式
- 操作数的选择
- 中断和异常

注意：输入/输出不包含在基本编程模型中。系统设计人员可以选择将I/O指令对应用可用或将这些功能留给操作系统。基于这个原因，I/O特性被放到了第II部分。

本章包含一节内容，正常情况下，那里的每项特性对于应用来说都是可见的。

2.1 存储器组织和段

物理存储器在80386系统下被组织成一个8位的字节序列。每个字节含有一个唯一的地址，从0到最大值2**32 -1（4G）。

然而，80386的程序与物理地址空间是相互独立的。这意味着写程序时可以不用关心有多少物理存储器，指令和数据在物理存储器中是如何存放的。

对应用程序可见的存储器组织模型是由系统软件设计者来决定的。80386的体系结构给予了设计者为每个任务选择一种模式的自由。存储器组织模型可以是下面几种：

- “平坦”地址空间是由最多4G字节组成的单个数组。
- 段地址空间可以由16,383个线性地址空间组成，每个4G。

两种模式均可以提供存储器保护。不同的任务可以使用不同的存储器组织模型。设计者使用存储器组织模型的准则和系统程序员使用什么方式来实现在第II部分－系统编程中讨论。

2.1.1 “平坦”模式

在“平坦”模式下，应用程序可以使用232（4G）字节的数组。尽管存储器可以是4G字节，但通常它们要小的多；处理器通过第5章介绍的地址变换将4G空间映射到物理存储器。应用程序不需要知道这些细节。

指向平坦地址空间的指针是一个32位序列数，从0到232 -1。单独编译模块的重定位由系统软件来做（例如，链接器，定位器，绑定器，加载器）。

2.1.2 段模式

在段模式下，应用程序可以使用更大的地址空间（称为逻辑地址空间）多达246（64T）字节的数组。尽管存储器可以是4G字节，但通常它们要小的多；处理器通过第5章介绍的地址变换将64T的地址空间映射到物理存储器（最多4G）。应用程序不需要知道这些细节。

应用程序可以把逻辑地址空间看作是16,383个一维子空间的集合，每个都有指定的长度。每个线性子空间被称作段。段是连续地址空间的一个单位。段大小可以从一个字节到最多232字节（4G）。

这个地址空间的一个完整指针由两部分组成（见图2－1）：

1．16位段选择符，标识一个段。

2．32位偏移，段内偏移地址。

在程序执行期间，处理器用段选择符和段的起始物理地址联系起来。单独编译的模块通过改变段基地址在运行时重定位。段大小是可变的；因此，段可以和它里面的模块大小相同。

2.2 数据类型

字节，字和双字是基本数据类型（参考图2－2）。字节开始于任何逻辑地址，由8位连续位组成。位被从0到7编号；位0是最低有效位。

字开始于任何逻辑地址，由连续的2个字节组成。因此字含有16位。位编号从0到15；位0是最低有效位。包含位0的字节称为低位字节；包含位15的字节为高位字节。

字中的每个字节都有自己的地址，较小的地址为字的起始地址。低位地址包含字的低8位，高位地址包含高8位。

双字开始于任何逻辑地址，由连续的2个字组成。因此字含有32位。位编号从0到31；位0是最低有效位。包含位0的字称为低位字；包含位31的字节为高位字。

双字中的每个字节都有自己的地址，最小的地址为双字的起始地址。含有最低位地址的字节包含8位最低位，位于最高地址处的字节包含8位最高位。图2－3展示了字和双字的布局。

注意：字不必分配在偶数地址，双字也不用在被4整除的地址上。这允许数据结构（例如，混合包含了字节，字和双字）在存储器的使用上可以有更多的自由和更高的效率。在32位总线配置下，双字在处理器和存储器之间的实际传输发生在被4整除的地址处；然而，处理器会将未对齐的字和双字转换为存储器接口可以接受的合适的请求序列。这种未对齐数据的传输会因为额外的存储器指令周期而降低性能。为了获得最高性能，数据结构（包括堆栈）应该以下面的方式来设计：无论何时，只要有可能，将字对齐在偶数地址处，双字对齐在被4整数的地址处。由于指令预取和队列，不要求指令在字和双字边界上。（不过，如果传输控制的目标地址能够被4整除将导致速度的微小提升。）

尽管字节，字和双字是操作数的基本类型，处理器同样也支持对其他操作数的解释。依赖于和操作数有关的指令，下面附加的数据类型可以被识别：

整型：

包含在32位双字，16位字，或8位字节中的有符号二进制数值。所有的操作数等于2的幂。符号位位于字节的位7，字的位15，双字的位31。符号位0为正值，1为负值。由于最高位用于符号位，8位整数的取值范围从-128到+127；16位整数从-32,768到+32,767；32位整数从-3231到+231-1。零的符号位为正。

序数：

包含在32位双字，16位字，或8位字节中的无符号二进制数值。所有位都被视为确定数字的量级。8位序数的取值范围从0-255；16位序数从0-65,536；32位序数从0-232 -1。

短指针：

32位逻辑地址。短指针是段内偏移。短指针用于平坦模式或段模式。

长指针：

48位的逻辑地址：16位段选择符和32位的偏移量。长指针只有系统设计者选择了段模式时才能使用。

字符串：

连续的字节，字或双字序列。字符串包含0-232 -1（4G）个字节。

位域：

连续的位序列。位域可以开始于字节的任何位置，最多32位。

位字符串：

连续的位序列。位域可以开始于字节的任何位置，最多232 -1位。

BCD码：

一个字节（未压缩），代表从0-9的十进制数。未压缩十进制数被存储为无符号字节数。每个字节存一个数。数字的大小由字节的低4位决定；十六进制值0-9是合法值，被解释为十进制数。高4位在乘法和除法中必须为零；加法和减法中可以含有任何值。

压缩BCD码：

一个字节（已压缩），代表两位十进制数，每位从0-9。每个十进制数存在字节的高/低4位中。高4位中数字为高数量级。0-9为合法值。压缩十进制字节取值范围从0-99。

图2－4用图形总结了80386支持的数据类型。

Figure 2-1. Two-Component Pointer

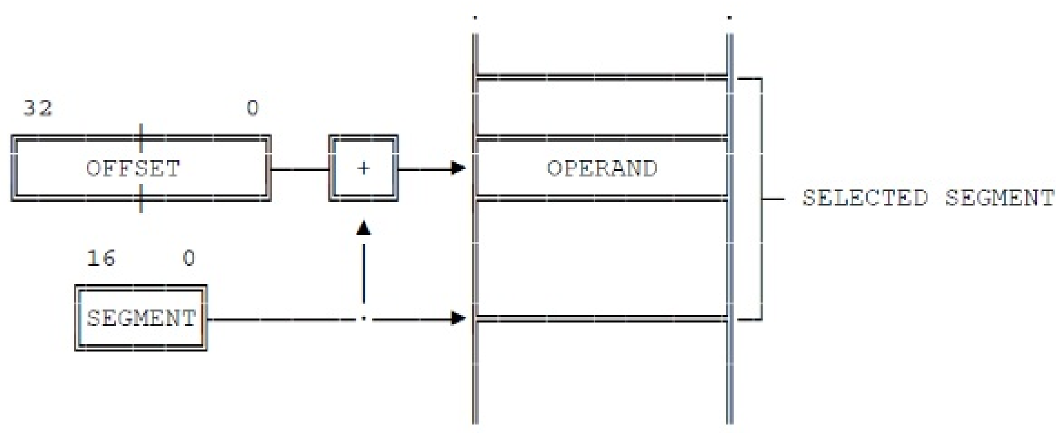


Figure 2-2. Fundamental Data Types

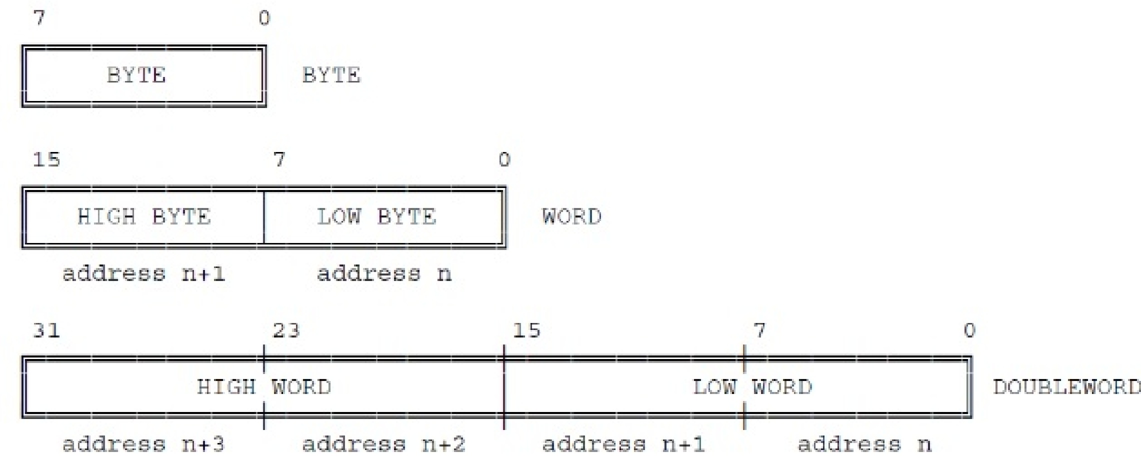


Figure 2-3. Bytes, Words, and Doublewords in Memory

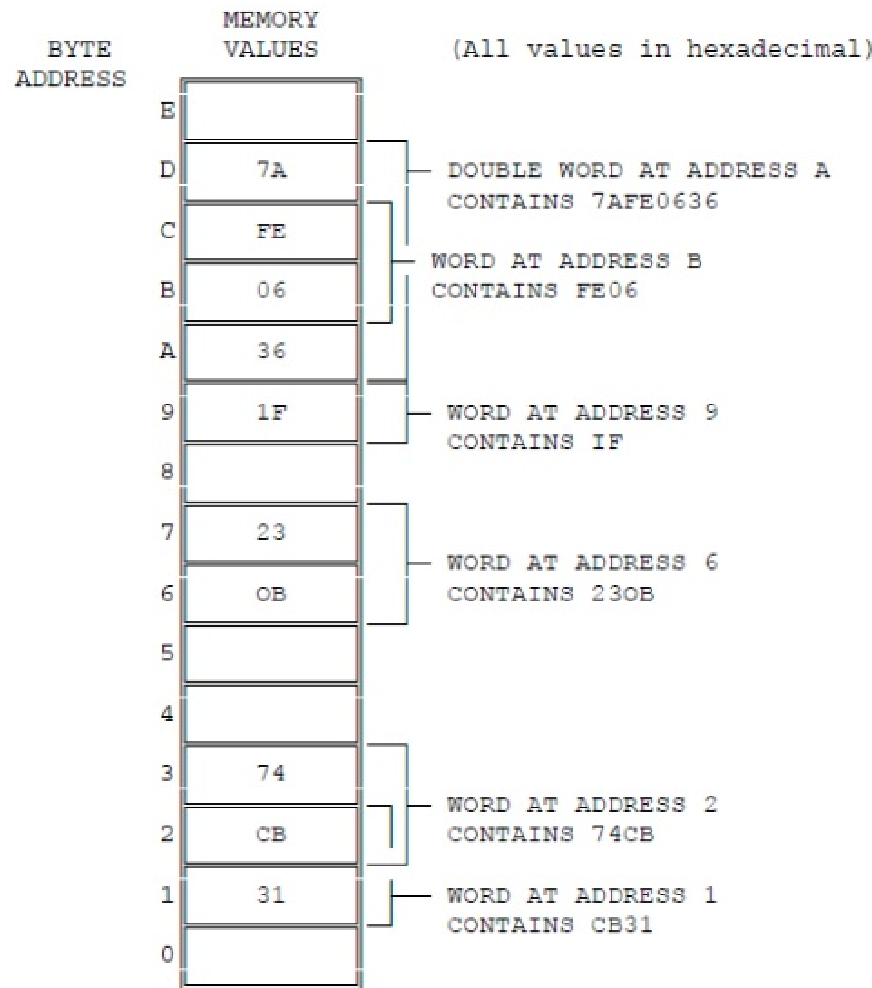
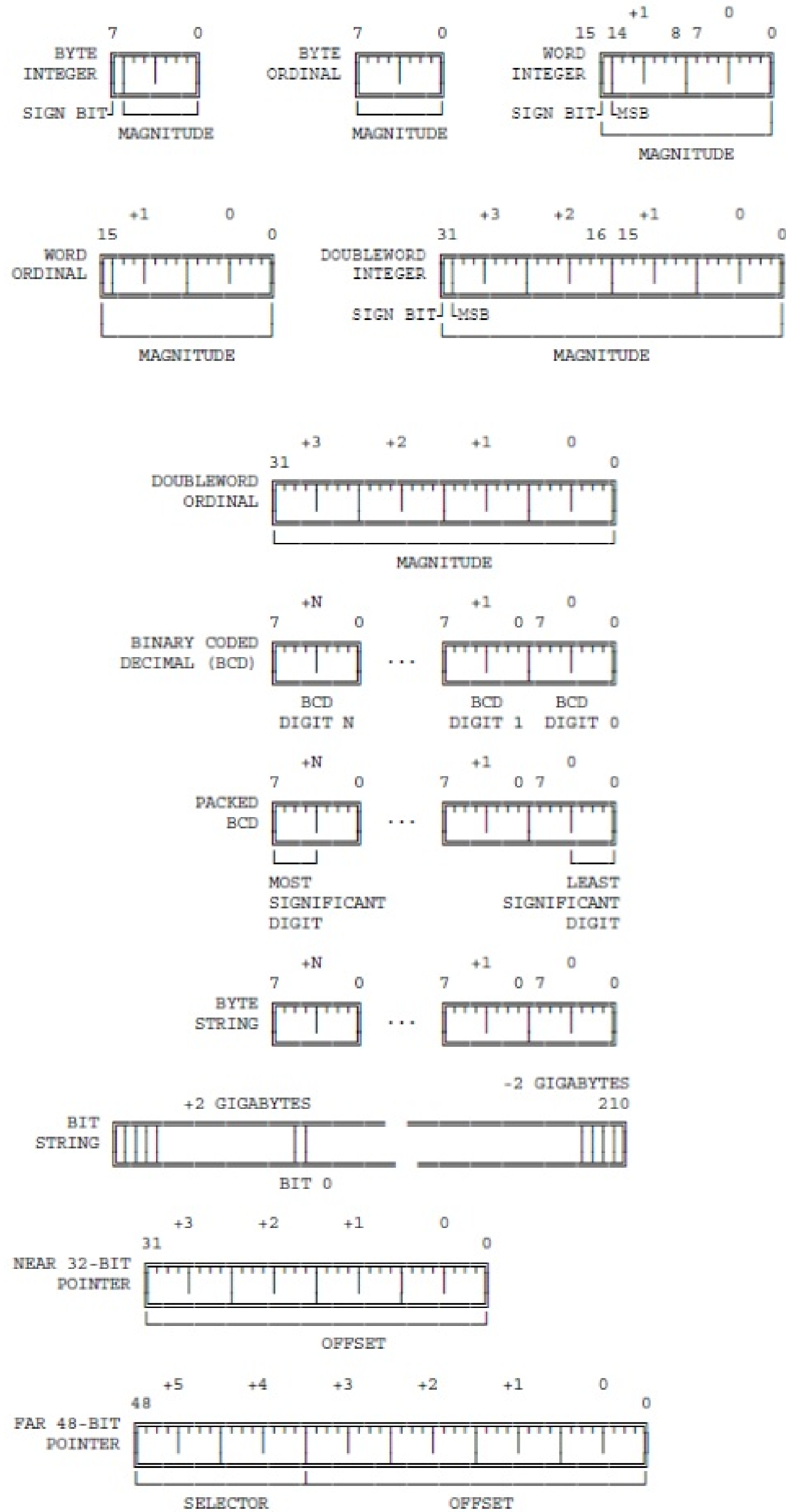
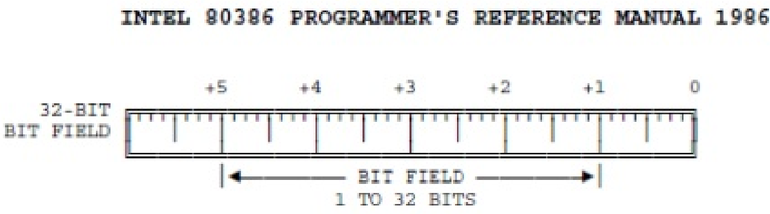


Figure 2-4. 80386 Data Types




INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986
+5
+4
+3
+2
+1
0
32-BIT
BIT FIELD
BIT FIELD
1 TO 32 BITS

2.3 寄存器

80386中应用程序员感兴趣的有16个寄存器。如图2－5所示，这些寄存器被分成以下几个基本类型：

1. 通用寄存器。这些32为通用寄存器主要用来数学和逻辑运算。 2. 段寄存器。这些特殊目的寄存器允许系统软件设计者选择平坦模式或是段模式。这六个寄存器决定了，任何时候，哪段存储器可以被寻址。 3. 状态和指令寄存器。这些特殊目的寄存器用于记录和改变80386处理器状态的一些特征。

2.3.1 通用寄存器

80386的32位通用寄存器包括EAX, EBX,ECX, EDX, EBP, ESP, ESI以及EDI。这些寄存器可以互换使用，存储逻辑和算术操作数。也可以互换地用于地址计算（有个例外，ESP不能被用作索引操作数）。

如图2－5所示，这8个寄存器中的低位字都有单独的名称，可以单独使用。这有利于处理16位数据项，以及和8086和80286保持兼容。字寄存器被命名为AX, BX, CX, DX, BP, SP, SI以及DI。

图2－5同样表明，16为寄存器AX, BX, CX和DX的每个字节都有单独的名称，可以独立使用。这有利于处理字符和8位数据项。字节寄存器被称为AH, BH, CH, DH（高位字节）；AL, BL, CL, DL（低位字节）。

所有这些寄存器均可以用来地址计算，作为大多数算术和逻辑计算的结果；然而，一些功能要求使用特定的寄存器。通过隐式的使用这些寄存器，80386架构可以使编码变得更紧凑。使用特定寄存器的指令包括：双精度乘法和除法，I/O，字符串指令，变换，循环，变量移位和循环，堆栈操作。

2.3.2 段寄存器

段寄存器给了系统软件设计人员在各种存储器组织模式之间选择的自由。存储器模式的实现是第II部分的主题－系统编程。设计人员可以选择一种模式，这种模式下，应用程序不需要改变段寄存器，在这种情况下，应用程序员可以跳过这章。

完整的程序通常包含许多不同的模块，每个由指令和数据构成。然而，在任何给定的程序执行时间段，只有一小部分程序模块的子集在使用。80386架构可以利用了这一点，它提供直接访问当前模块环境的手段，在有需要时访问其他段。

在任何给定的时间，六个存储器段可以在程序执行期间被立即访问。段寄存器CS, DS, SS, ES, FS和GS用来标识这六个当前段。这些寄存器每个都表示一个特殊类型的段，就像图2－6中关联助记符（”code”, “data”, 或者”stack”）表示的那样。每个寄存器唯一地确定一个特殊

段，这些段组成了程序，在稍后以最快的速度被立即访问。

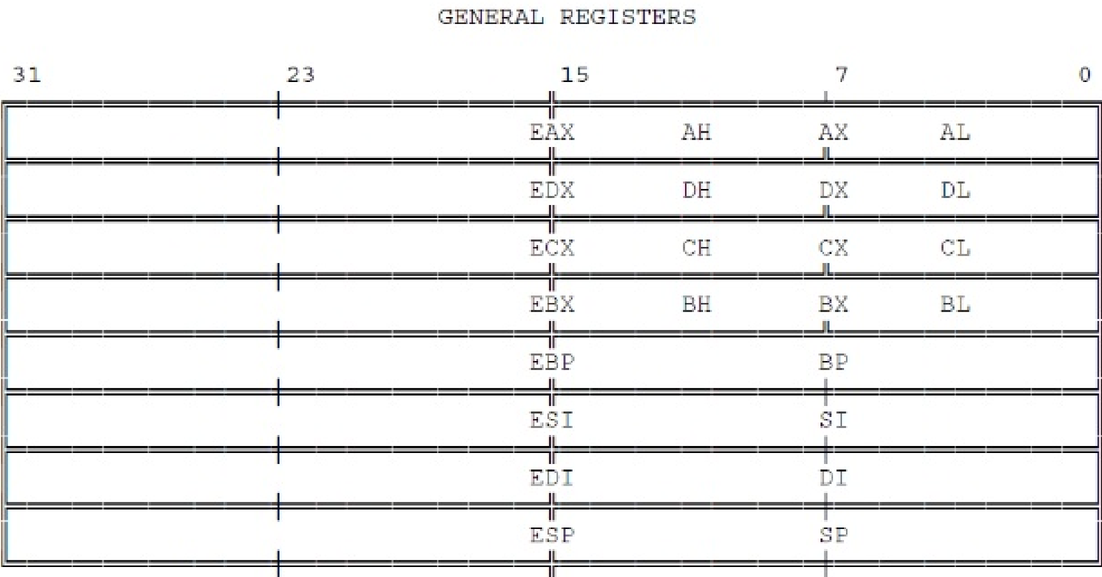
包含当前指令执行序列的段称为当前代码段；它通过CS寄存器来声明。80386将指令指针的内容作为偏移量，从这个段来提取所有当前指令。CS寄存器在段内控制传输指令执行（例如，CALL和JMP）后被隐式的改变。

子程序，参数和程序活动记录通常要求在堆栈上分配一块存储空间。所有堆栈操作都用SS来定位堆栈。与CS不同，SS可以被显示的加载，从而允许程序动态的定义堆栈。

DS, ES, FS和GS寄存器允许声明4个数据段，每个都可以被正在执行的程序寻址。访问4个独立的数据区帮助程序高效的访问不同的数据类型；例如，一个数据段寄存器指向当前模块的数据结构，另一个执行上层模块导出的数据，另一个指向动态创建的数据，而另一个指向和其他任务共享的数据。段内的操作数可以在指令中声明偏移量或通过通用寄存器来间接访问。

取决于数据结构（比如说，数据被放到一个或更多段内的方式），一个程序可能需要访问多于4个数据段。要想访问额外的段，DS, ES, FS和GS可以在程序执行期间在控制下改变。只需要在访问数据的指令前面执行一条指令来加载合适的段寄存器。

处理器会把一个基地址和每个由段寄存器选择的段联系起来。要想寻址段内的数据项，32位的偏移量被加到段基址上。一旦选择了一个段（通过加载段选择符到段寄存器），数据操作指令只需要声明偏移量。如果只声明了偏移量，使用一个简单的规则来确定使用哪个段寄存器。



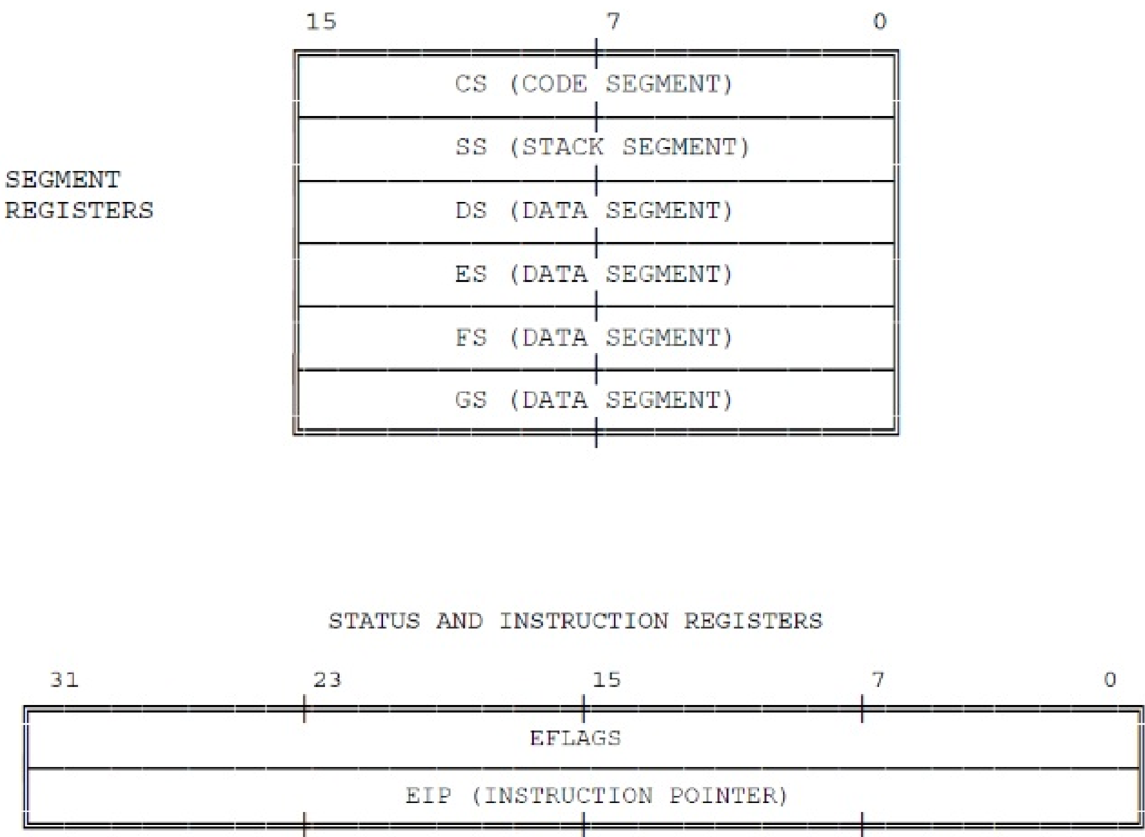
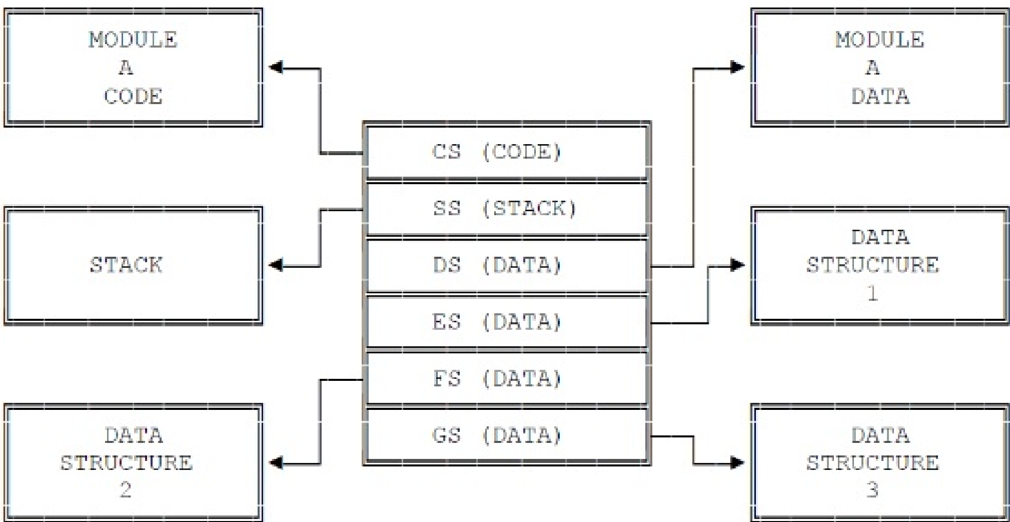


Figure 2-6. Use of Memory Segmentation



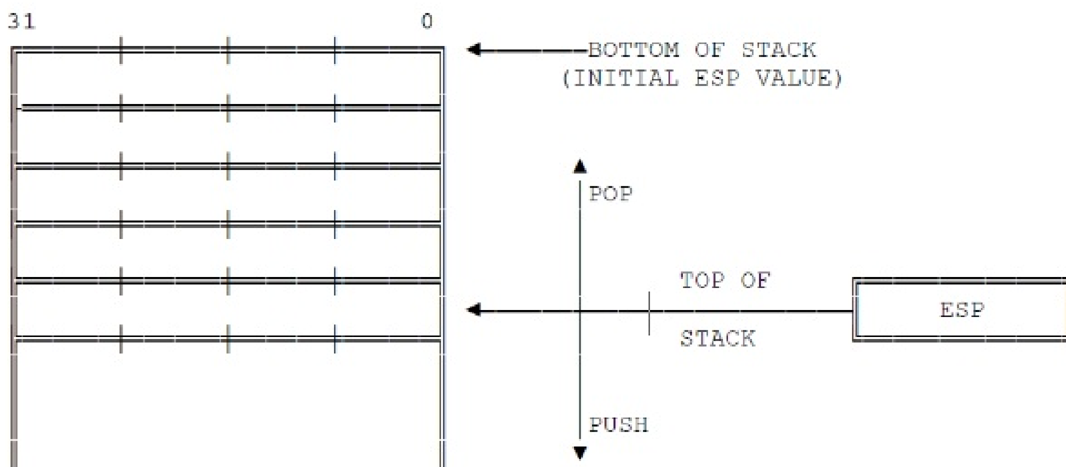
2.3.3 堆栈的实现

堆栈操作通过下面的三个寄存器变得容易些：

1. 堆栈段寄存器（SS）。堆栈的实现在存储器中。系统可以有多个堆栈，只受限于堆栈总数的上限值。堆栈可以有4G空间，堆栈的最大长度。在同一时刻只能有一个堆栈可以访问－SS定位的那个。这个是当前堆栈，常被简称为“这个“堆栈。处理器的所有堆栈操作自动使用

SS。 2. 堆栈指针寄存器（ESP）。ESP指向向下增长堆栈（TOS）的顶部。ESP被下面的操作间接引用：PUSH和POP操作，子程序调用和返回，以及中断操作。当数据被放入堆栈时（见图2－7），处理器减小ESP，然后将数据写入新的TOS。当数据从堆栈弹出时，处理器从TOS中拷贝数据，然后增加ESP。换句话说，堆栈在内存中项低位地址方向增长。 3. 堆栈帧基指针（EBP）。EBP是访问堆栈中的数据结构，变量和动态分配的工作空间的最好选择。EBP经常通过相对堆栈的一个固定参考点而不是当前TOS来访问数据项。它的典型用法是标识为当前进程建立的当前堆栈帧的基地址。当EBP用作偏移计算的基址寄存器是，偏移量自动在当前堆栈段内（即，由SS选择的当前段）计算。因为SS不用显示的声明，这种编码指令更有效。EBP也可以用作是通过其他寻址段寄存器的索引。

Figure 2-7. 80386 Stack

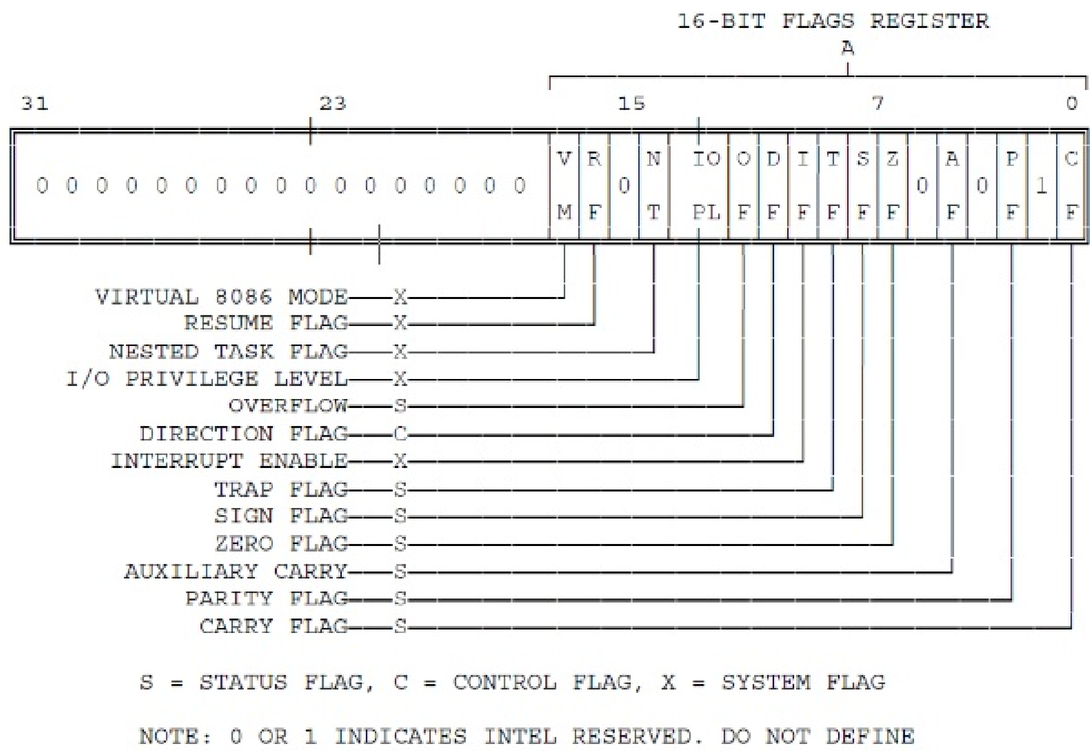


2.3.4 标志位寄存器

标志位寄存器为32为寄存器，命名为EFLAGS。图2－8定义了寄存器中位。这些标志空着特定的操作并表示80386的状态。

EFLAGS的低16位被命名为FLAGS，可以单独使用。该特性在执行8086和80286代码时非常有用，因为EFLAGS的这部分和8086和80286的FLAGS寄存器是一样的。

标志位可以分为3组：状态标志位，控制标志位，以及系统标志位。系统标志位的讨论推迟到第II部分。



2.3.4.1 状态标志位

EFLAGS寄存器的状态标志位允许一条指令的结果影响下一条指令。算术指令使用OF, SF, ZF, AF, PF和CF。SCAS（扫描字符串），CMPS（比较字符串），以及LOOP指令使用ZF来通知它们的动作已结束。有些指令可以在算术指令执行之前设置，清除以及取反CF。每个状态标志位的定义参见附录C。

2.3.4.2 控制标志位

EFLAGS的控制标志位DF控制着字符串指令。

DF（方向标志位，位10）

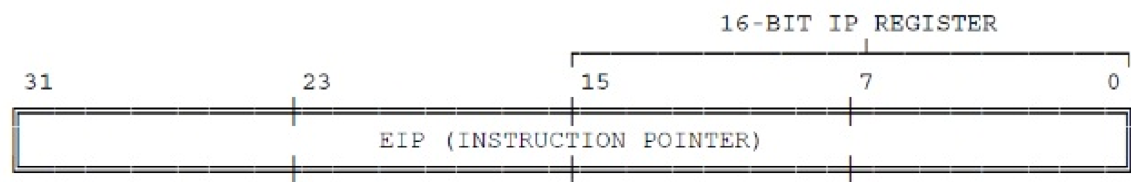
设置DF标志位使字符串指令自动递减；也就是，从高位地址到低位地址处理字符串。清除DF使字符串指令自动递增，从低位地址到高位地址处理字符串。

2.3.4.3 指令指针

指令指针寄存器（EIP）包含相对于当前代码段内下一个将执行的指令序列的地址偏移量。对于程序员来说，指令指针不是直观可见的；它被控制传输指令，中断以及异常隐式控制。

如图2－9所示，EIP的低位16位被命名位IP，可以单独使用。该特性对于执行位8086额80286设计的指令非常有用。

Figure 2-9. Instruction Pointer Register



2.4 指令格式

80386指令编码信息包括操作的声明，操作数的类型以及操作数的位置。如果操作数位于存储器中，指令必须选择，显示的或隐式的，哪个当前可寻址段包含这些操作数。

80386指令由各种要素构成并有着不同的格式。指令的详尽描述在附录B（译者注：附录A?）；指令要素在下面描述。在这些指令要素中，只有一个，操作码，是必须的。其他的要素依据涉及到的特定操作和操作数的类型和位置可有可无。指令要素，以出现的顺序描述如下：

- 前缀－一个或多个在指令前面的字节，修改指令的动作。应用程序可以使用下面几种前缀：

  1. 段重载－显示的声明指令使用哪个段寄存器，从而覆盖80386为指令使用的默认段寄存器。 2. 地址大小－在32位和16位地址之间切换。 3. 操作数大小－在32位和16位操作数之间切换。 4. 重复－用在字符串指令，使指令作用于字符串的每一项。

- 操作码－声明指令的执行动作。有些操作有不同的操作码，每个声明一个不同的操作。

- 寄存器声明符－一条指令可以声明1到2个寄存器操作数。寄存器声明符可以出现在相同的位置作为指令码，或作为地址模式声明符。

- 地址模式声明符－当有这项时，它用来声明操作数是寄存器还是存储器位置；如果位于存储器，声明是否要使用移位，基址寄存器，索引寄存器，以及缩放。

- SIB（scale, index, base）字节－当地址模式声明符表明要使用索引寄存器来计算操作数地址是，SIB字节被包含在指令中，来编码基地址寄存器，索引寄存器以及缩放因子。

- 移位－当地址模式声明符表明要使用移位来计算操作数地址时，移位被编码在指令中。移位是一个32位，16位或8位整数。在通常情况下，当移位足够小时使用8位形式的移位。处理器扩展8位移位到16或32位，考虑符号位。

- 立即数－当有这项时，它直接给出了操作数的值。立即数可以时8，16，32位宽。当8位操作数以某种方式和16位和32位数联合使用时，处理器自动扩展8位操作数，考虑符号位。

2.5 操作数选择

一条指令可以有零或多个操作数－指令操作的数据。零操作数的一个例子是NOP（no operation）。操作数可以在下面的位置：

- 位于指令本身（立即数）
- 位于寄存器（EAX, EBX, ECX, EDX, ESI, EDI, ESP, 或者EBP，如果是32位操作数；AX, BX, CX, DX, SI, DI, SP, 或者BP，如果是16位操作数；AH, AL, BH, BL, CH, CL, DH, 或者DL，如果是8位操作数；段寄存器；位操作的EFLAGS寄存器）
- 位于存储器
- 位于I/O端口

立即数和寄存器操作可以比存储器操作数有更快的访问速度，因为存储器操作数必须从存储器中取出来。寄存器操作数可以从CPU中获得。立即数同样从CPU获得，因为它们被作为指令的一部分而被预取。

对于那些带有操作数的指令来说，有些隐式的声明操作数；有些显式的声明操作数；其他的使用隐式和显式两种方式的组合来声明操作数；例如：

隐式操作数：AAM

根据定义，AAM（ASCII adjust for multiplication）操作AX寄存器。

显示操作数：XCHG EAX, EBX

要交换的操作数被编码在指令中操作码后面的位置。

隐式和显式操作数：PUSH COUNTER

存储器变量COUNTER（显式操作数）被拷贝到堆栈（隐式操作数）的顶部。

注意：大部分指令都有隐式操作数。例如，所有的算术指令更新EFLAGS寄存器。

80386指令可以显式的引用1到2个操作数。2个操作数的指令，如MOV, ADD, XOR等，通常用结果覆盖其中一个参与操作数。因此可以区分处源操作数（不受操作影响的一个）和目的操作数（被结果覆盖的一个）。

对于多数指令，两个显式指定的操作数的一个－或者是源，或者是目的操作数－可以位于寄存器或者存储器。另一个操作数必须是寄存器或者是立即数作为源操作数。因此，显式2个操作数指令允许操作数有如下几种：

- 寄存器到寄存器
- 寄存器到存储器

- 存储器到寄存器
- 立即数到寄存器
- 立即数到存储器

然而，一些字符串指令和堆栈操作指令从存储器到存储器传输数据。有些字符串指令隐式的声明操作数，并且都位于存储器。入栈和出栈操作允许在存储器和基于存储器的堆栈之间传输数据。

2.5.1 立即数

一些指令使用指令自身的一部分数据作为一个（有时是两个）操作数。这样的操作数被称作是立即数。操作数可以是32位，16位，或者8位长。例如：

```
SHR PATTERN, 2
```

指令的一个字节含有数值2，变量PATTERN的移位数。

```
TEST PATTERN, 0FFFF00FFH
```

指令中的双字包含要测试变量PATTERN的掩码。

2.5.2 寄存器操作数

操作数可以被放置在下面32位通用寄存器之一（EAX, EBX, ECX, EDX, ESI, EDI, ESP, 或者EBP），16位通用寄存器之一（AX, BX, CX, DX, SI, DI, SP, 或者BP），8位通用寄存器之一（AH, BH, CH, DH, AL, BL, CL, 或者DL）。

80386有引用段寄存器的指令（CS, DS, ES, SS, FS, GS）。只有在系统设计人员选择了段模式的时候，应用程序才能使用这些指令。

80386有些指令用来引用标志寄存器。标志位可以存储在堆栈中，并从堆栈中恢复。一些指令会直接改变在EFLAGS中通常会被修改的标志位。其他标志位很少被修改，它们可以通过堆栈中的标志位镜像而被间接的改变。

2.5.3 存储器操作数

寻址存储器操作数的数据操作指令必须声明（直接或间接的）包含操作数的段以及操作数在段内的偏移量。然而，为了指令编码的速度和紧凑，段选择符被存储在高速段寄存器内。因此，如果要寻址一个存储器操作数，数据操作指令只需要声明要求的段寄存器和偏移量。

寻址存储器操作数的80386数据操作指令使用下面方法之一来声明操作数在段内的偏移量：

1. 大多数访问存储器的数据操作指令包含一个字节，显式的指明操作数的寻址方式。一字节，被认为是modR/M字节，跟在操作码后面，指明操作数位于寄存器还是存储器中。如果操作数在存储器中，地址由一个段寄存器和下面的任意值计算出来：基址寄存器，索引寄存器，缩放因子，移位。当使用索引寄存器时，modR/M字节跟在标识索引寄存器和缩放因子的字节之后，这种寻址方式具有最高的自由度。

2. 一些数据操作指令隐式的使用特殊寻址方式：

- 对于一些隐式使用EAX寄存器的短型MOV，操作数的偏移量被编码为双字放入指令中。没有使用基址寄存器，索引寄存器以及缩放因子。
- 字符串操作隐式的使用DS:ESI, (MOVS, CMPS, OUTS, LODS, SCAS) 或者ES:EDI (MOVS, CMPS, INS, STOS)来寻址。
- 堆栈操作隐式的通过SS:ESP寄存器来寻址；例如，PUSH, POP, PUSHA, POPA, POPAD, PUSHF, PUSHFD, POPF, POPFD, CALL, RET, IRET, IRETD, 异常以及中断。

2.5.3.1 段选择

数据操作指令不需要显式声明使用哪个段寄存器。对于所有这些指令，段寄存器的声明是可选的。对于所有的存储器访问，如果没有显式的声明一个段，处理器根据表2－1的规则来自动选择一个段寄存器。（如果系统设计人员已经选择了平坦模式，那么段寄存器和处理器选择它们的规则对于应用程序来说不是很明显）。

存储器引用的类型和操作数驻留的段之间有着紧密的联系。通常，引用存储器暗示着当前段为数据段（即，暗示着段选择符在DS中）。然而，ESP和EBP用来访问堆栈中的数据项；因此，当ESP和EBP作为基址寄存器使用时，当前段为堆栈段（即，SS含有选择符）。

特殊的指令前缀要素可以覆盖默认的段选择。段重载前缀允许显式的段选取。80386对于每个段寄存器都有一个段重载前缀。只有在下面的特殊情况下，隐式的段选择才不会被重载：

- 在字符串指令中用于目的字符串的ES。
- 堆栈指令中SS的使用。
- 取指令时CS的使用。

表2－1. 缺省段寄存器选择规则

需要存储器引用	使用的段寄存器	隐式段选择规则
指令	Code（CS）	自动指令预取
堆栈	Stack（SS）	所有堆栈的入栈和出栈。任何使用ESP或者EBP作为基址寄存器的存储器引用。
本地数据	Data（DS）	除了堆栈和目的字符串的所有数据引用。
目的字符串	Extra（ES）	字符串指令的目的操作数。

2.5.3.2 有效地址计算

modR/M字节为寻址方法提供了最大的自由度，需要modR/M作为第二个字节的指令在80386指令集中非常常见。对于由modR/M定义的存储器操作来说，段内偏移通过对下面三部分求和计算出来：

- 指令中的移位。
- 基址寄存器。
- 索引寄存器。索引寄存器可能会自动与缩放因子2，4，或者8相乘。

上面生成的结果称为有效地址。构成有效地址的每个成员可以时正值，也可以是负值。如果所有成员的和超过了232，有效地址会被截短为32位值。图2－10展示了modR/M寻址的所有可能。

移位，由于是编码在指令中，所以作固定运算很有用；例如：

- 简单缩放操作数的位置。
- 静态分配数组的开始。
- 记录中一项的偏移量。

基址和索引具有类似的功能。都使用相同的通用寄存器集合。都可以用来计算地址中需要动态确定的部分；例如：

- 进程参数的位置以及堆栈中的局部变量。
- 在几个相同记录类型的记录或记录数组中，其中一个的开始。
- 一位数组或多维数组的开始。
- 动态分配数组的开始。

当通用寄存器用于基址或索引时，它们有下面的不同：

- ESP不能用作索引寄存器。
- 当ESP或者EBP用于基址寄存器时，缺省段由SS指定。所有其他情况，使用DS作为段选择器。

缩放因子允许数组项是2，4，或8字节宽时用索引高效的访问数组。索引寄存器的移位在寻址时由处理器完成，不会损失性能。也避免了单独的移位或乘法指令。

基址，索引，和移位这三个部件可以任意组合；任何一个上面的部件可以是空的。缩放因子只有在使用了索引因子时才能用。每种可能的组合对于由高级语言程序员和汇编程序员使用的数据结构非常有用。下面时一些寻址部件不同组合后的可能使用：

移位

单独的移位只是操作数的偏移量。该部件用于直接寻址静态分配的缩放操作数。可以使用8位，16位或32位移位。

基址

操作数的偏移量由通用寄存器的一个间接声明，作为“基”变量。

基址 + 移位

寄存器和移位的组合可以用于两种不同的目的：

1. 对中元素大小不是2，4，或8字节的静态数组进行索引。移位编码为相对于数组开始的偏移量。寄存器内保存计算结果，决定一项指定的元素在数组内的偏移量。

2. 访问记录中的一项。移位定位记录中的数据项。寄存器选择出现的记录中的一个，因此为这种常见的功能提供了一种紧凑的编码。

这种组合一个重要的特殊情形就是访问在堆栈中的进程活动记录的参数。这种情况下，EBP是作为基址寄存器的最好选择，因为当EBP被用作基址寄存器时，存储器自动使用堆栈段寄存器（SS）来定位操作数，因此为这种常见的功能提供了一种紧凑的编码。

（索引 + 缩放） + 移位

当静态数组中元素大小是2，4，或者8时，这种组合提供了有效的索引。移位定位在数组的开始，索引寄存器保存要求的数组元素的下标，处理器自动应用缩放因子将下标转换为索引。

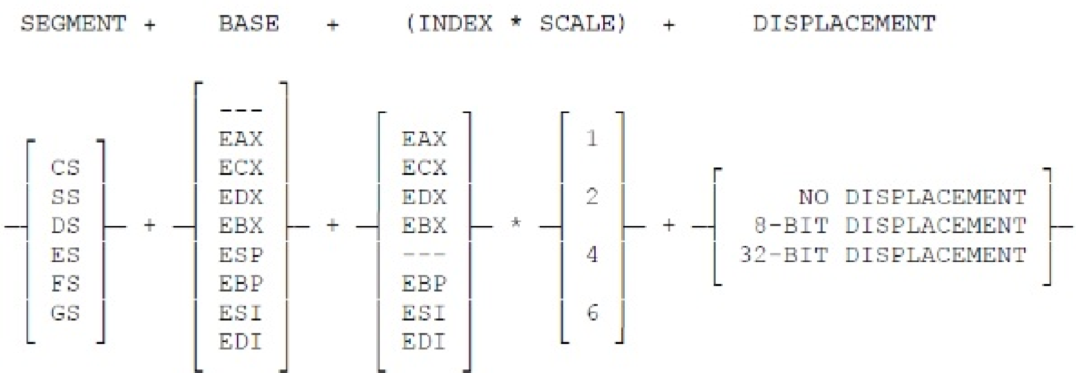
基址 + 索引 + 移位

两个寄存器加在一起支持二维数组（移位决定数组的开始）或者记录数组的一个实例（移位指示记录中的一项）。

基址 + (索引 * 缩放) + 移位

当数组中元素是2，4，或者8字节宽时，这种组合提供了一种二维数组的高效索引。

Figure 2-10. Effective Address Computation



2.6 中断和异常

80386有两种手段来中断程序的执行：

1. 异常是同步事件，当CPU在指令执行期间检测到某些条件时作出的反映。

2. 中断是异步事件，通常由需要引起注意的外部设备触发。

中断和异常有一点是相同的：它们都需要CPU暂停正在执行的程序，转去执行更高优先级的程序。这两种中断的主要区别在于它们的发生源。异常在重新执行引起异常的程序和数据时总能复现，而中断通常独立于当前正在执行的程序。

正常情况下，应用程序不关心中断的处理。系统程序员可以在第9章找到更多中断的信息。然而，应用程序员对有些异常更感兴趣，许多操作系统给予了应用程序处理异常的机会。不过，操作系统自己定义应用程序和80386异常机制之间的接口。

表2－2高亮显式了那些应用程序感兴趣的异常。

- 当DIV或IDIV的分母为零或商对于目的操作数太大均产生除数为零异常。（DIV和IDIV的讨论参见第3章。）
- 从陷阱标志位（TF）产生的调试异常将重返应用程序。
- 当执行INT 3后产生中断点异常。该指令被调试器用来在指定地点中断程序的执行。
- 当执行INTO指令或OF（overflow）标志被置位（在算术操作后，置位OF标志）时，产生溢出异常。（有关INTO的讨论参见第3章）。
- 当执行BOUND指令或数组索引超出数组边界时，产生边界检查异常。（有关BOUND指令的讨论参见第3章。）
- 非法操作码在一些应用中用来扩展指令集。这种情况下，非法指令异常的产生让我们有机会模拟操作码。
- 当程序中用到了协处理器指令，但系统中却没有协处理器时，产生“协处理器不可得”异常。
- 当协处理器检测到非法操作时，产生协处理器错误。

INT指令在任何时候执行时都会产生中断；处理器把这个中断按照异常来处理。这个中断的作用（以及所有其他异常）取决于应用程序提供的异常处理程序，或者作为系统软件的一部分（由系统程序提供）。INT指令本身在第3章讨论。有关异常的完整讨论参见第9章。

表2－2 80386保留异常和中断

向量号	描述
0	除数错误
1	调试异常
2	不可屏蔽（NMI）中断
3	中断点
4	INTO检测溢出
5	BOUND越界
6	非法操作码
7	协处理器不可得
8	双精度异常
9	协处理器段溢出
10	非法任务状态段
11	段缺失
12	堆栈错误
13	通用保护
14	页错误
15	（保留）
16	协处理器错误
17-32	（保留）

第4章 系统寄存器

80386 的许多结构特性都只能被系统程序员所使用。这一章将对它的系统结构做一个概述。

80386 系统级特性包括以下几点：

内存管理

保护机制

多任务

输入/输出

中断和异常

初始化过程

协处理器和多处理器

调试

这些特性是通过寄存器和指令实现的，以下几节将对它们进行介绍。这一章的目的只是对以后几章的一个概述，而不是对所有的细节进行描述。这一章所讲到的寄存器、指令将会做一些解释，或者在以后的章节中对其进行详细的说明。

4.1 系统寄存器 (System Registers)

为系统程序员设计的寄存器可以分为以下几类：

EFLAGS （标志寄存器）

Memory-Management Registers （内存管理寄存器）

Control Registers （控制寄存器）

Debug Registers （调试寄存器）

Test Registers （测试寄存器）

4.1.1 系统标志 (System Flags)

系统标志寄存器EFLAGS 控制着 I/O、可屏蔽中断（maskable interupts）、调试(debuging)、任务切换(task switching)、保护模式下虚拟8086方式的执行、多作务环境(multitasking environment)。这些标志在图 4-1 中被高亮显示。

IF （ 中断许可标志 Interrupt-Enable Flag，比特位 9）

设置 IF 使CPU可识别外部（可屏蔽）中断请求。复位 IF 则禁止中断。 IF 对不可屏蔽外部中断和异常的识别没有任何作用。关于中断的详细信息，请参看第9章的描述。

NT （嵌套任务 Nested Task， 比特位 14）

处理器用嵌套位来控制被中断或被调用的任务链。NT 对 IRET 指令的操作有影响。更多的信息请参看第7章和第9章。

RF （继续位 Resume Flag, 比特位 16）

RF 位暂时禁止调试异常，以便一条指令可以在一个调试异常结束后立即重看书而且不会引发另一个调试异常。参看第12章以获得更多的细节。

TF （陷阱位 Trap Flag，比特位 8）

设置 TF 可以让处理器工作在单步调试模式。在此模式下，CPU 每执行完一条指令后将自动引发一个异常，这样可以在程序每执行完一条指令后对程序进行查询。单步仅仅是80386 众多调试特性的一个。参看第12章以获得更多的细节。

VM （虚拟8086 模式 Virtual 8086 Mode，比特位 17）

当此位设置时，VM 标志说明一个任务正在执行一个8086 程序。参看第14章以得到更多80386 在保护模式下执行8086 任务、多任务环境的详细介绍。

4.1.2 内存管理寄存器 (Memory –Management Registers)

80386 有4个寄存器来寻址特定的数据结构，它们用来实现段式内存管理。

GDTR 全局描述符表寄存器 (Global Descriptor Table Register)

LDTR 局部描述符表寄存器 (Local Descriptor Table Register)

这些寄存器指向段描述符表 GDT 和 LDT。第5章对通过描述符表来寻址的机制做了详细的介绍。

IDTR 中断描述符表寄存器 (Interrupt Descriptor Table Register)

这个寄存器指向一张包含中断处理子程序入口点的表（IDT）。第9章对中断机制进行的详细介绍。

TR 任务寄存器 (Task Register)

这个寄存器指向当前任务信息存放处，这些信息是处理器所需要的。第7章对80386的多任务特性做了介绍。

4.1.3 控制寄存器 (Control Registers)

图4-2显示了80386的控制寄存器，CR0、CR2、和CR3。这些寄存器可以通过MOV 指令的一些变种形式被系统程序员所访问，这样便可以把它们存入通用寄存器或从通用寄存器中加载，例如：

```
MOV      EAX ,     CR0
MOV      CR3 ,     EBX
```

CR0 包含系统控制标志，这些标志控制着整个系统的运行，而不仅仅是针对某一个特定的任务。

EM （摸拟位 Emulation，比特位 2）

EM 指示协处理器功能是否通过摸拟来实现。更多的信息请参看第11章。

ET （扩展类型 Extension Type， 比特位 4）

ET 指明了系统内协处理器的类型（80287 或80387）。详细情况请查看第11章和第10章。

MP （数学部件存在 Math Present，比特位 1）

MP 控制 WAIT 指令的执行，WAIT 用于系统与协处理器的同步。第11章对其进行详细介绍。

PE （保护模式允许 Protection Enable，比特位 0）

设置PE 将让处理器工作在保护模式下。复位PE将返回到实模式工作。关于模式切换请参看第14章和第10章。

PG （分页允许 Paging， 比特位 31）

PG 指明处理器是否通过页表来转换线性地址到物理地址。关于分页地址转换请查看第5章。关于如何设置PG位，请查看第10章。

TS （任务已切换Task Switched，比特位 3）

处理器第次做任务切换时将设置 TS 位， 当执行协处理器指令时将会测试 TS 位。详细信息请查看第11章。

CR2 被用来当PG位置位时，处理缺页异常。当发生缺页异常时，处理器自动将引起缺页异常的线性地址存放到CR2。关于缺页中断请查看第9章。

CR3 只有当PG 位设置时才有用。通过CR3，CPU 可以定位当前任务的页目录表。关于页表和页地址转换机制请查看第5章。



4.1.4 调试寄存器 (Debug Register)

调试寄存器使80386有很好的调试功能，包括断点、不改变代码段情况下设置指令断点。关于它们的格式和用途请参看第12章。

4.1.5 测试寄存器 (Test Registers)

测试寄存器并不是80386体系结构的标准部件。它们仅仅是用来测试TLB 地址转换信息，这些存贮的是来自页表中的。查看第12章，关于怎样使用这些寄存器。

4.2 系统指令 (System Instructions)

系统指令能完成以下功能：

1、 检测指针参数 (Verification of pointer parameters)（参看第6章）：

ARPL —— 调整 RPL (Adjust RPL)

LAR —— 加载访问权限 (Load Access Rights)

LSL —— 加载段界限 (Load Segment Limit)

VERR —— 读检验 (Verify for Reading)

VERW —— 写检验 (Verify for Writing)

2、 寻址描述符表 (Addressing descriptor tables)（参看第5章）：

LLDT —— 加载局部描述符表寄存器 (Load LDT Register)

SLDT —— 存储局部描述符表寄存器 (Store LDT Register)

LGDT —— 加载全局描述符表寄存器 (Load GDT Register)

SGDT —— 存储全局描述符表寄存器 (Store GDT Register)

3、 多任务 (Multitasking) (参看第7章)：

LTR —— 加载任务寄存器 (Load Task Register)

STR —— 存储任务寄存器 (Store Task Register)

4、 协处理器和多处理器 (Coprocessing and Multiprocessing) (参看第11章)：

CLTS —— 清除任务已切换标志 (Clear Task-Switched Flag)

ESC —— 转译指令 (Escape instructions)

WAIT —— 等待直到协处理器空闲 (Wait until Coprocessor not Busy)

LOCK —— 引发总线锁信号 (Assert Bus-Lock Signal)

5、 输入和输出 (Input and Output) (参看第8章)：

IN —— 输入

OUT —— 输出

INS —— 输入串

OUTS —— 输出串

6、 中断控制 (Interrupt control) (参看第9章)：

CLI —— 清除中断允许标志位 (Clear Interrupt-Enable Flag)

STI —— 设置不断允许标志位 (Set Interrupt-Enable Flag)

LIDT —— 加载中断描述符表寄存器 (Load IDT Register)

SIDT —— 存储中断描述符表寄存器 (Store IDT Register)

7、 调试 (Debugging) (参看第12章)：

MOV —— 向调试寄存器输入或输出 (Move to and from debug registers)

8、 TLB 测试（TLB testing）（参看第10章）：

MOV —— 向测试寄存器输入或输出 (Move to and from test registers)

9、 系统控制 （System Control）：

SMSW —— 保存机器状态字（Set MSW）

LMSW —— 加载机器状态字（Load MSW）

HLT —— 处理器挂起（HALT Processor）

MOV —— 向控制寄存器输入或输出（Move to and from control registers）

SMSW 和 LMSW 指令主要用于兼容80286 处理器。80386 程序可以通过变形的MOV 指令访问CR0，来访问MSW。 HLT 指令使处理器停止工作，直到收到了个 INTR 或者 RESET 信号。

除了在上面提到的章节外，每条指令还可以在我们推荐的章——第17章，中找到相关介绍。

第五章 内存管理

80386转换逻辑地址（也就是，程序员观点的地址）到物理地址（也就是，实际的物理内存地址）分以下两步：

1、 分段地址转换，这一步中把逻辑地址（由段选择子和段偏移组成）转换为线性地址。

2、 分页地址转换，这一步中把线性地址转换为物理地址。这一步是可选的，由系统软件设计者决定是否需要。

这些转换对于应用程序员来说是不可见的。图5-11以高度抽象的形式显示了这两步转换。

图5-11和这一章的以下部分以一种简单的方式介绍了80386的地址转换机制。事实上，地址转换机制也包括了内存保护的特性。为了简单起见，保护机制将放另一章，第六章来讲述。



5.1 分段地址转换（Segment Translation）

图5-2详细显示了处理器如何把逻辑地址转换为线性地址。

为了这样的转换，处理器用到了以下的数据结构：

1、 描述符（Descriptors）

2、 描述符表（Descriptor tables）

3、 选择子（Selectors）

4、 段寄存器（Segment Registers）

5.1.1 描述符（Descriptors）

段描述符是处理器用来把逻辑地址映射为线性地址的必要数据结构。描述符是由编译器、连接器、加载器、或者是操作系统生成的，不能由应用程序员生成。图5-3显示了两种常用的描述符的格式。所有的段描述符都是这两种格式当中的一种。段描述符的字段是以下：

基址（BASE）：决定了一个段在4G线性地址空间中的位置。处理器把3部分基址联接在一起，来形成一个32位的基址。

界限（LIMIT）：决定了一个段的大小。处理器用2部分的界限字段来形成一个20位的界限值。处理器以两种方式来解析界限的值，解析方式取决于粒度位的设置情况：

1、 当单元大小为一个字节时，则定义了一个最大为1M字节的段。

2、 如果单元大小为4K字节，则段大小可以高达4G。界限值在使用之前处理器将会把它先左移12位，低12位则自动插入0。

粒度位（Granularity bit）：决定了界限值被处理器解析的方式。当它被复位时，界限值被解析为以1字节为一个单元。当它置位时，则界限值以4K为一个单元。

类型（TYPE）：用于区别不同类型的描述符。

描述符特权级（Descriptor Privilege Level）（DPL）：用来实现保护机制（参看第六章）。

段存在位（Segment-Present bit）：如果这一位为0，则此描述符为非法的，不能被用来实现地址转换。如果一个非法描述符被加载进一个段寄存器，处理器会立即产生异常。图5-4显示了当存在位为0时，描述符的格式。操作系统可以任意的使用被标识为可用（AVAILABLE）的位。一个实现基于段的虚拟内存的操作系统可以在以下情况下来清除存在位：

1、 当这个段的线性地址空间并没有完全被分页系统映射到物理地址空间时。

2、 当段根本没有在内存里时。

已访问位（Accessed bit）：当处理器访问该段时，将自动设置访问位。也就是说，当一个指向该段描述符的选择子被加载进一个段寄存器时或者当被一条选择子测试指令使用时。在段级基础上实现虚拟内存的操作系统可能会周期性的测试和清除该位，从而监视一个段的使用情况。

创建和维拟描述符是系统软件的任务，一般说来可能是由，编译器、程序加载器、系统生成器、或者操作系统来协作完成的。





5.1.2 描述符表（Descriptor Tables）

段描述符存储在以下两种描述符表当中的一个：

1、 全局描述符表（GDT）

2、 一个局部描述符表（LDT）

就象图5-5所示的一样，一个描述符表仅仅是一个包含了很多描述符的8字节内存数组而以。描述符表是长度是可变的，最多可包含高达8192（2^13）个描述符。便是处理器是不会使用全局描述符表的第一项（INDEX=0）的。

处理器用GDTR和LDTR来定位内存中的全局描述符表和当前的局部描述符表。这些寄存器存储了这些表的线性地址的基址和段长界限。指令LGDT和SGDT是用业访问全局描述符表寄存器的，而指令LLDT和SLDT则是用来访问局部描述符表寄存器的。





5.1.3 选择子（Selectors）

线性地址部分的选择子是用来选择哪个描述符表和在该表中索引一个描述符的。选择子可以做为指针变量的一部分，从而对应用程序员是可见的，但是一般是由连接加载器来设置的。图5-6显示了选择子的格式。

索引（Index）：在描述符表中从8192个描述符中选择一个描述符。处理器自动将这个索引值乘以8（描述符的长度），再加上描述符表的基址来索引描述符表，从而选出一个合适的描述符。

表指示位（Table Indicator）：选择应该访问哪一个描述符表。0代表应该访问全局描述符表（GDT），1代表应该访问局部描述符表。

请求特权级（Requested Privilege Level）：保护机制使用该位（参看第六章）。

由于全局描述符表的第一项是不被处理器使用的，所以当一个选择子的索引（Index）部分和表指示位（Table Indicator）都为0的时候（也就是说，选择子指向全局描述符表的第一项时），可以当做一个空的选择子。当一个段寄存器被加载一个空选择子时，处理器并不会产生一个异常。但是，当用一个空选择子去访问内存时，则会产生异常。这个特点可以用来初始化不用的段寄存器，以防偶然性的非法访问。

Figure 5-6. Format of a Selector

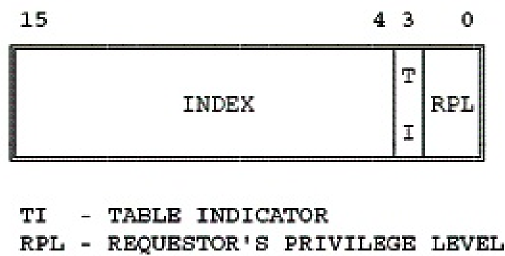


Figure 5-7. Segment Registers

	16-BIT VISIBLE SELECTOR	HIDDEN DESCRIPTOR
CS		
SS		
DS		
ES		
FS		
GS		

5.1.4 段寄存器（Segment Registers）

80386把描述符的信息存储在段寄存器里，以便不用每次内存访问都去访问内存中的描述符表。

如图5-7所示，每一个段寄存器都有一个可见部分和一个不可见部分。这些段寄存器的可见部分被程序员当作一个16位的寄存器来使用。不可见的部分则只能由处理器来操纵。

加载这些寄存器的操作和一般的加载指令是一样的（和第三章讲述的相同），这些指令分为两类：

1、 真接的加载指令，例如，MOV，POP，LDS，LSS，LGS，LFS。这些指令显示的访问这些段寄存器。

2、 隐式的加载指令，例如，far CALL和JMP。这些指令隐式的访问CS 段寄存器，将它加载一个新的值。

使用这些指令，程序将用一个16位的选择子加载段寄存器的可见部分。处理器将自动的将基址、界限、类型、和其它信息从描述符表中加载到段选择子的不可见部分。

因为很多数据访问的指令都是访问一个已加载段寄存器的数据段，所以处理器可以用与段相关的基址部分加上指令提供的偏移部分，而且不会有额处的加法开销。

5.2 分页地址转换（Page Translation）

在地址转换的第二个阶段，80386将线性地址转换为实物理地址。这个阶段实现了基于页的虚拟内存和页级保护机制。

分页地址转换过程是可选的。只有当CR0中的PG位置位时才会产生效果。这个位的设置一般来说是由操作系统在系统初始化的过程中设置的。如果操作系统想要实现能运行多个虚拟8086任务、基于页级的保护、基于页级的虚拟内存的话，PG位是必需置位的。

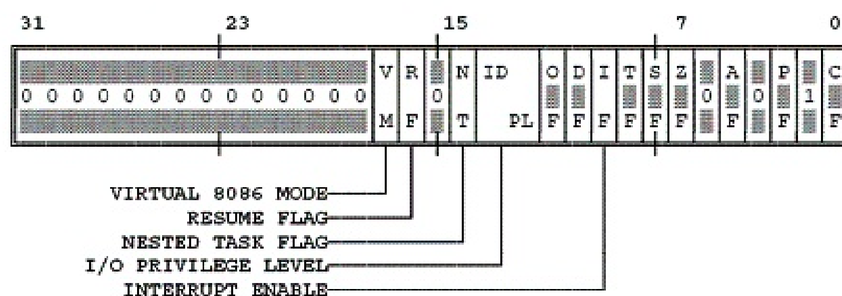
5.2.1 页桢（Page Frame）

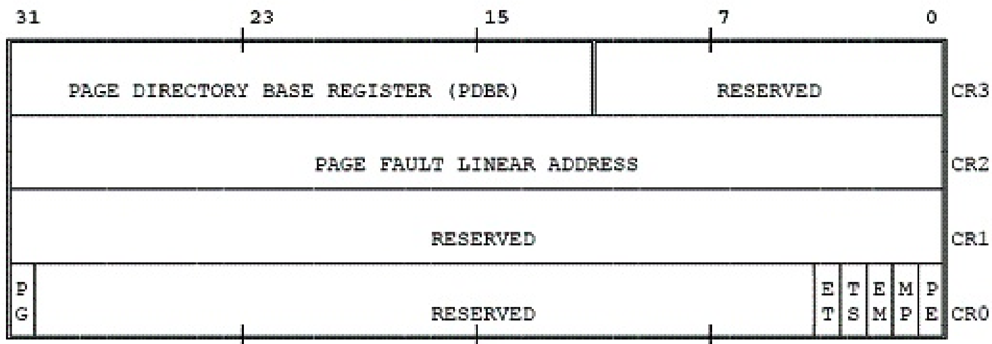
一个页桢是一个地址连续的4K大小单元内存。各页以字节边界为起始，大小固定不变。

5.2.2 线性地址（Linear Address）

一个线性地址间接的访问到一个实物理地址外。它通过使用一个页表，表内的一个页，和一个页内的偏移来映射到实物理地址外。图5-8显示了线性地址的格。

图5-9显示了处理器如何将线性地址中的DIR，PAGE，和OFFSET字段转换为实物理地址上的，这个过程使用了两级页表。寻址机制使用DIR字段来索引页目录表，用PAGE字段来索引页表，这样就可以确定一个物理页桢了，然后再使用OFFSET部分来索引该物理页桢，最终访问所需要的数据。





5.2.3 页表（Page Tables）

一个页表仅仅是很多32-位页指示器（32-bit page specifiers）组成的一个数组。页表本身也是一个页，所以包含了4K字节内存空量（最多可包含1K个32-位的表项）。

在寻址一个内存页时，使用了两级的页表。高一级的页表也被叫作页目录。页目录可最多寻址1K个二级页表。一个二级页表最多可寻址1K个页面。所以，一个页目录最多可寻址1M个页面。因为每个页面有4K（2^12）字节大小。所以一个页目录可寻址整个80386的实物理地址空间（2^20 * 2^12 = 2^32）。

5.2.4 页表项（Page-Table Entries）

两级页表项都有相同的格式，图5-10显示了这种格式。

5.2.4..1 页桢地址（Page Frame Address）

页桢地址指出了一个实物理页的开始地址。因为页的地址是以4K为边界的，所以地址的低12位总是为0。在页目录中，页桢地址是二级页表的起始地址。在二级页表中，页桢地址是所要要访问的物理页的起始地址，该物理页包含了要访问的指令操作数。

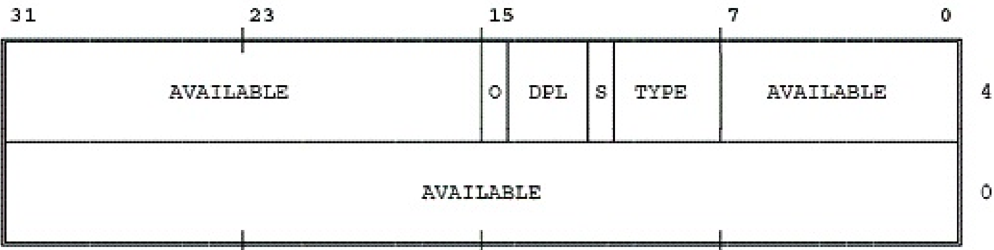
5.2.4.2 存在位（Present Bit）

存在位决定了一个页表项是否可以用作地址转换过程，如果P=1则可以用该页表项。

当任何一级页表项的P=0时，该项都不可以用作地址转换过程，这时，该项的其它位可以被软件使用。它们中的任何一位都不会被硬件使用。图5-11显示了当P=0时的页表项格式。

当任何一级页表项的P=0时，而软件又试图用它来访问内存时，处理器将会引发一个异常。在支持页级虚拟内存的软件系里，缺页异常处理子程序可以将所需的页面调入物理内存。引起缺页异常的指令是可以重起的，关于异常处理的更多信息请参看第9章。

注意，没有页目录自身的存在位。当任务挂起时，该任务的页目录是可以不存在的，但是操作系必须在一个任务被重运行前确保该任务的CR3映象（保存在TSS里）指示的页面（即页目录表）在内存中。关于TSS和任务指派的详细信息请参看第7章。





5.2.4.3 已访问位和脏位（Accessed and Dirty Bits）

这些位提供了两级页表的数据使用情况信息。除了页目录表的脏位（Dirty bit），所有的这些位都由硬件自动置位，但是处理器绝对不会复位它们。

在一个页面被读或写之前，处理器将自动将两级页表的这些相关的位置1。

当向一个地址写入时，处理器将会把相关的二级页表的脏位（Dirty bit）置为1。页目录表项的脏位没有作定义。

当系统内存紧张时，一个支持页级虚拟内存的操作系统可以使用这些位来决定将要换出哪些物理页面。操作系统应该自已负责测试和清除这些相关位。

参看第11章，学习80386如何在多处理器环境下更改访问位和脏位。

5.2.4.4 读/写 位，用户/特权用户 位（Read / Write and User / Supervisor Bits）

这些位并不是用于地址转换过程的，它们是用来实现页级保护机制的，这些保护机制是在地址转换过程的同时实施的。参看第六章，以了解理多关于保拟机制特性。

5.2.5 页地址转换缓存（Page Translation Cache）

为了获得最大的地址转换效率，处理器把最近使用的页表数据存储在一个芯片内的缓存中。只有当所要的地址转换信息没有在缓存中时，才有访问两级页表的必要。

应用程序员是感觉不到页地址转换缓存的存在的，但系统程序员知来说不是。当页表内容改变时，操作系统程序员必须清除缓存。页地址转换缓存可以用以下两种方法清除：

1、通过MOV 指令重新加载CR3寄存器，例如，MOV CR3， EAX。

2、通过任务切换到一个TSS，该TSS保存了一个不同的CR3映象。关于任务切换，请查看第7章。

5.3 混合分段和分页地址转换（Combining Segment and Page Translation）

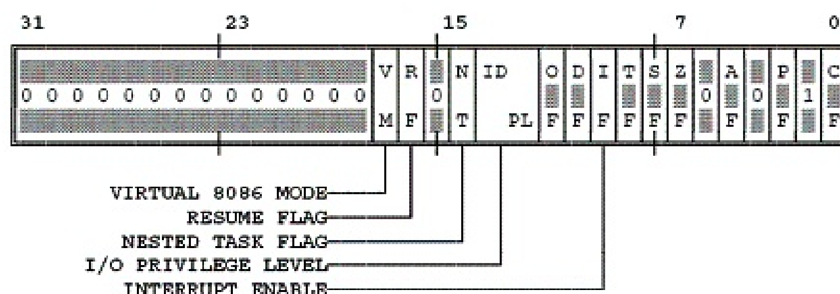
图5-12 结合了图5-2和图5-9来对两阶段（从逻辑地址到线性，再从线性地址到实物理地址（当启用分页时））的地址转换做一个总结。通过使用不同的方法，内存管理软件可以实现几种不同形式的内存管理机制。

5.3.1 “平坦”体系结构（”Flat Architecture”）

当80386用来执行一些程序，而这些程序也为别的不支持分段的处理器而设计时，有效的“关闭”分段可能比较好。80386没有禁止分段的执行模式，但是同样的效果是可以通过一些特定的方法实现的：把指向包括整个32-位地址空间的描述符的选择子加载到段寄存器里，段选择子没有必要改变。32-位的偏移已足够寻址整个80386支持的内存空间了。

5.3.2 跨多个页的段（Segments Spanning Several Pages）

80386系统结构允许一个段比内存页（4K）大，也允许比内存页小。比如，有一个段用来寻址和保护一个大小为132K的数据结构。在一个支持页级虚拟内存的软件系统里，没有必要把这一整个段都调入实物理内存。该结构被分成功33个页面，任何一个都可以不存在。应用程序员不会感觉到虚拟内存系统在以这种方式调动页面。



5.3.3 跨段的页面（Pages Spaning Several Segments）

在另一方面，段可能比一个页面要小。比如，考虑一个数据结构（如信号量（Semaphore））。因为段的保护和共享机制，把每一个信号量放在一个段里也许比较好些。但是，由于一个系需要很多的信号量，如果为每一个信号量分配一页的话效率很低下。所以，把几个段合并到一个页面里应该更好。

5.3.4 非对齐的页和段边界（Non-Aligned Page and Segment Boundaries）

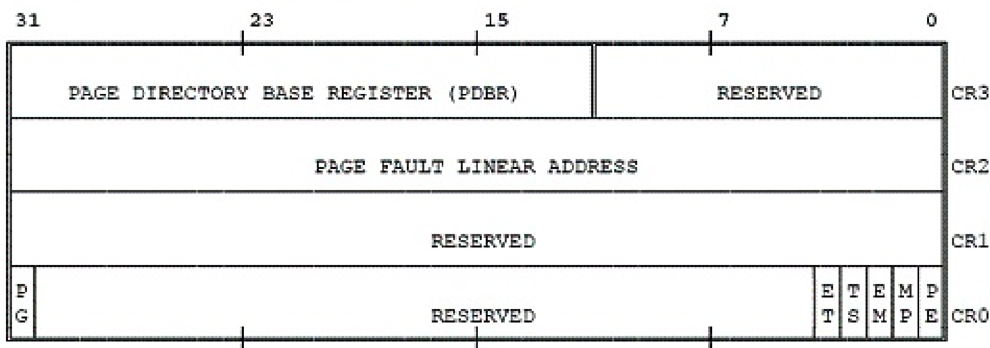
80386系统并不强求页和段的任何对齐。即使一个页包含了一个段的结尾又包含了一个段的开始也是完全可以的。类似的，即包含一个页的开始和另一个页的结尾的段也是完全允许的。

5.3.5对齐的页和段边界（Aligned Page and Segment Boundaries）

如果页与段之间有一定的对齐的话，对于内存管理系统来说也许会简单很多。例如，如果一个段只以页为单元来分配的话，段页逻辑将会结合起来。就没有为部分页面而管理的逻辑了。

5.3.6 每段一个别页表（Page-Table Per Segment）

一个更简单的内存空间管理方法便是将每一个段对应为一个页目录项，图5-13显示了这种方式。每个描述符的基址部分的低22位都将是0。换言之，基址被映射到每个页表的第一项。每个段长度可以从1到4M的任意大小。一个段可以包含1到1K个物理内存页，多少则由长度界限字段来决定。这样的话，一个任务可以寻址1K个段（对于很多应用程序来说都足够了），每个段可以高达4M字节。描述符，和与之对应的页目录项，还有与之对应的页表，就可以同时分配同时回收。



第六章 内存管理

6.1 为什么要保护（Why Protection?）

80386保护机制有助于找出和调试程序的BUG。80386支持很复杂的程序，它们可以包含成百上千的程序模块。在这样的程序中，关键问题是如何快而有效的找出程序的BUG，并将之造成的损坏降到最小。为了能让程序便于调试和生成高质量的产品，80386包含了检验内存访问、指令执行等很多保护机制。根据系统设计者的设计目标，这些保护机制可以使用，也可以忽略。

6.2 80386保护机制概述（Overview of 80386 Protection Mechnaisms）

80386的保护机制主要包括以下几方面：

1、 类型检查（Type Checking）

2、 界限检查（Limit Checking）

3、 可寻址空间约束（Restriction of addressable domain）

4、 子程序入口点约束（Restriction of procedure entry points）

5、 指令集约束（Restriction of instruction set）

内存管理硬件也被集成到了80386的硬件保护机制当中。保护机制同时施于分段地址转换过程和分页地址转换过程。

每一次的内存访问都将被检查，以保证没有违反保护机制。所有这些检测都是在内存访问周期之前执行的。所有的违规行为都将引发异常。因为检测是与地址转换同时进行的，所以并没有性能上的损失。

非法的内存访问将引发异常。关于异常更多的信息，请查阅第9章。这一章只介绍引发异常的非法操作。

“特权级”（”privilege”）的概念是很多保护机制的核心。对于子程序，特权级是指一个子程序被信赖的程度，这种信赖程度可以使别的子程序或数据免受损害。对于数据，特权级是指对数据结构的保护程度，这种程度可以让该数据结构免受不信任代码的访问。

特权级的概念对分段机制和分页机机制同时有效。

6.3 段级保护（Segment-Level Protection）

段保机护机制有以下五个方面：

1、 类型检查（Type Checking）

2、 界限检查（Limit Checking）

3、 寻址范围约束（Restriction of addressable domain）

4、 子程序入口点约束（Restriction of procedure entry points）

5、 指令集约束（Restriction of instruction set）

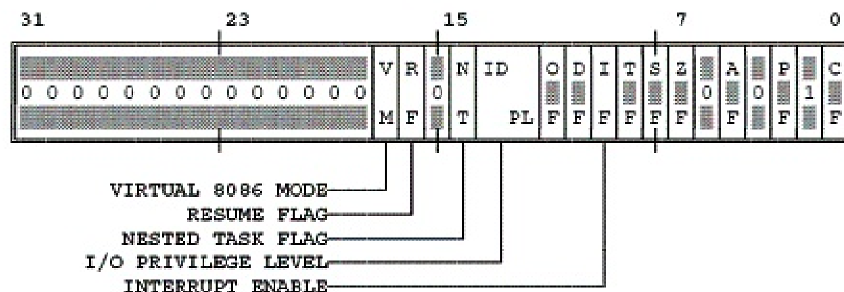
段是保护的单元，段描述符用来存储保护机制参数。当把一个选择子加载进段描述符时和每次段访问时CPU自动执行保护检查。段选择寄存器保存着当前可寻址段的保护机制参数。

6.3.1s 描述符存储保护机制参数（Descriptors Store Protection Parameters）

图6-1高亮显示了段描述符中与保护相关的字段。

在描述符创造时，系统软件同时把保护参数入在描述符中。一般来说，应用程序员不用管保护参数的。

当程序把一个选择子装入段寄存器时，处理器不仅加载段的基址部分，而且也把保护参数装入段寄存器。每个段寄存器有一个不可见的部分用来存放基址、界限、类型、和特权级。所以对于以后的保护检查，处理器不必浪费多于的时钟周期去从内存中加载这些信息。



6.3.1.1 类型检查（Type Checking）

描述符的类型字段有2个作用：

1、 它用来区分不同格式的描述符。

2、 它暗示了描述符的用处。

除了被应用程序广范使用的数据段和可执行段描述符外，80386还有特殊的描述符，用来描述和操作系统相关的段（Segment）和门（Gate）。图6-1列出了所有类型的系统段和门。注意，不是所有的描述符都定义了一个段。门描述符有不同的用法，下一章将讲述。

数据段和可执行段的类型字段包括了以下这些位，用来定义一个段的用途（参看图6-1）：

- 在一个数据段描述符中，可写位指出了指令是否有权向这个段写入数据。
- 在一个可执行段描述符中，可读位指出了指令是否有权从这个段中读出数据（例如，用来访问与指令一起储存的常量数据）。可读的可执行段可以在以下两种方式下读取：

1、 通过使用CS前缀，来访问CS 寄存器指定的段。

2、 把描述符加载到数据段寄存器（DS，ES， FS， GS）。

类型检查可以检测出某些程序错误，比如，当程序员访问一个不是为某种目的设定的段时。处理器在以下两种情况下检查类型：

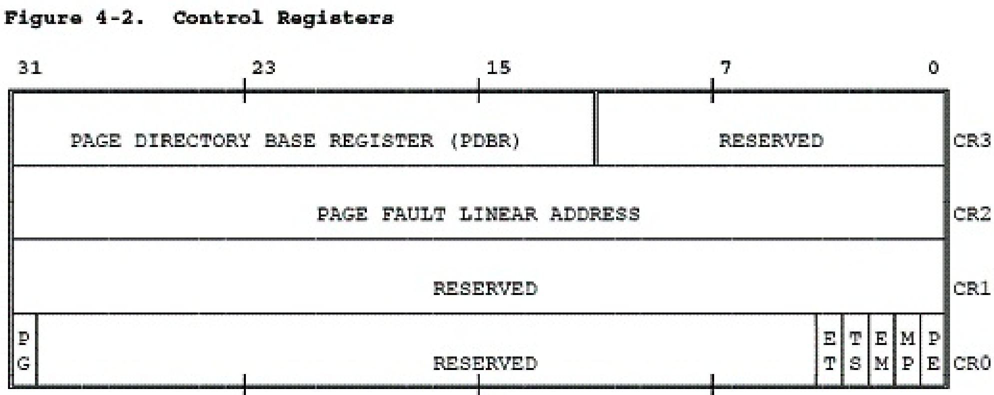
1、 当把一个描述符加载到一个段寄存器时。有些段寄存器只能加载某些类型的描述符，比如：

- CS 寄存器只能加载一个可执行段的描述符。
- 只有可执行的数据段描述符才能加载入SS段寄存器。

2、 当一条指令访问一个段时（显式的或者隐式的）。一些段只能通过一些特定的方式才能使用，例如：

- 可执行段不允许任何指令写入数据。
- 如果一个数据段的可写位没有置位，任何指令不可向其写入数据。
- 如果一个可执行段的可读位没有置位，任何指令不可从该段读取数据。

表6-1，系统段描述符和门描述符



6.3.1.2 界限检查（Limit Checking）

一个描述符的界限字段是用来防止程序在访问一个段时超出段的范围的。处理器根据描述符的G位（granularity bit）来解析界限字段的。对于数据段，处理器在解析界限字段时还要根据E位（expansion-direction bit）和 B位（big bit）（参看表6-2）。

当G=0 时，界限字段的值即是描述符中20位的 limit-field。这时，界限可能从0~0FFFFF（2^20 – 1 或者说 1 M）。当G=1时，处理器将会自动的在描述符的 limit-field 低位加12位0。这样，实际的界限值可以从0FFFH（2^12 – 1 或者说4K）到0FFFFFFFFH（2 ^ 32 – 1 或者说4G）。

除了向下延伸的段外，界限值总比段的大小少1（字节表示）。当以下任一情况发生时，处理器引发异常：

- 试图访问一个 地址 > 界限 的字节。
- 试图访问一个 地址 >= 界限 的字。
- 试图访问一个 地址 >= （界限 – 2 ） 的字双字。

对于向下延伸的数据段，界限做用相同，但是被处理器以不同的方式来解析。这个时候，有效地址则从 limit + 1 到 64K 或者 2^32 – 1（4G）(由B位决定)。向下延伸的段当界限设为0时，有最大的段长。

向下延伸的特性允许把堆栈拷贝到一个更大的段，而不改变内部段指针，来增大一个堆栈的大小。

描述符表的界限字段用来防止程序寻址超出一个描述符表。界限用来确定描述符表的最后一个描述符的最后一个字节。因为一个描述符是8字节长，界限字段的值为

N * 8 – 1 ,对于一个包含N个描述符的描述符表。

界限字段可以查测到类似下标出界和非法指针运算等程序错误。这些错误一当发生时就可以被发现，所以确定这种错误是很简单的。如果没有界限检查，这些的错误会使一个模块受到破坏，这种错误只有当下一次受损的模块不正常工作时才会被发现，而且发现也是比效困难的。

Table 6-2. Useful Combinations of E, G, and B Bits

Case:	1	2	3	4
Expansion Direction	U	U	D	D
G-bit	0	1	0	1
B-bit	X	X	0	1
Lower bound is:				
0	X	X		
LIMIT+1			X	
shl(LIMIT,12,1)+1				X
Upper bound is:				
LIMIT	X			
shl(LIMIT,12,1)		X		
64K-1			X	
4G-1				X
Max seg size is:				
64K	X			
64K-1		X		
4G-4K			X	
4G				X
Min seg size is:				
0	X	X		
4K			X	X

shl (X, 12, 1) = shift X left by 12 bits inserting one-bits on the right

6.3.1.3 特权级

处理器通过赋给一个重要的对象以一个 0 ~ 3 的数字来实现特权级。这个数字被称为特权级。0代表最高特权级，3代表最低特权级。以下的对象包含了特权级：

- 描述符包含了一个叫做描述符特权级（DPL）的字段。
- 选择子包含了一个叫做请求特权级（RPL）的字段。RPL 代表着指向子程序的选择子。
- 处理器的一个内部寄存器记录了一个叫做当前特权级（CPL）的字段。一般来说，CPL和当前正在执行的代码段的DPL是相同的。当控制在不同特权级的段间转移时，CPL发生变化。

当某个段的一个子程序要访问一个段时，处理器会自动的把一个要访问某个段的子程序的特权级和CPL或者更多的特权级相比。这种比较是在当一个描述符被加载到一个段寄存器时执行的。比较的标准在访问数据时和控制转移时是分别不同的。所以，就有了以下两种不同的查测：

图6-2显示了不同特权级环的解析方式。中心是用来放最关键的软件的，一般来说是操作系统内核。外面是用来放次关键的应用软件段的。

4个特权级全用并不是必要的。已存在的软件如果是主两级特权级设计的，也可以很好的被80386支持的。一个只用一级特权级的系统应该使用特权级0；一个使用两级特权级的系统应该使用特权级0和特权级3。



6.3.2 访问数据约束（Restricting Access to Data）

为了寻址一个操作数，80386必须把一个选择子加载到一个段寄存器（DS， ES， FS，GS， SS）里。处理器自动执行访问特权级的检查。检查是在当选择子被加载入段寄存器时执行的。图6-3显示了，在这种检测下的3种不同的特权级。

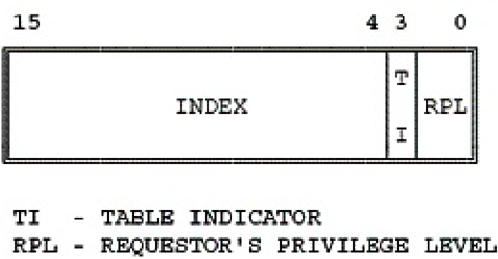
1、 CPL（当前特权级（current privilege level））

2、 用来指定目标段的选择子的RPL（请求特权级（requestor’s privilege level））。

3、 目标段的DPL（描述符特权级）

一条指令只有当一个目标段的DPL在数值上大于或等于CPL和选择子的RPL中的最大值时，才可以加载目标段的选择子到一个段寄存器里。也就是说，一个子程只能访问它同级的或比它特权级低的数据。

当一个任务的CPL改变时，它的可寻址范围也会改变。当CPL是0时，任何特权级的数据段都是可寻址的。当CPL是1时，只有特权级从1 ~ 3 的数据段才是可寻址的，当CPL是3时，只有特权级是3 的数据段才是可寻址的。80386的这个性质可以保护操作系的内部表不被应用程序所读取或更改。



6.3.2.1 在代码段中访问数据（Accessing Data in Code Segments）

更少见点的情况可能是在一个代码段内存储数据。代码段可以存储常量。任何指令不能向一个代码段写入数据。以下是可以在代码段内访问数据的方法：

1、 用一个非一致性的、可读的、可执行的选择子加载一个数据段寄存器。

2、 用一个一致性的、可读的、可执行的选择子加载一个数据段寄存器。

3、 用CS前缀来读取一个可读的，可执行的代码段（该段当前已被CS寄存器所指向）。

在访问数据时，和正常的数据访问规则适用于第1种情况。情况2总是合法的，一致性段的特权级总是和当前特权级（CPL）相同，无论该段的DPL是多少。第三种情况也是合法的，因为目标段的DPL就是代码段的DPL，根据定义，也就是CPL。

6.3.3 控制转移约束（Restricting Control Transfers）

在80386中，控制转移是通过指令 JMP, CALL, RET, INT, 和 IRET 当然还有异常和中断机制。异常和中断是特殊的情况，在第9章中讲述。这一章讲述JMP，CALL，和RET 指令。

JMP, CALL, RET 指令的“NEAR”（近）形式，只在当前的代码段内发生转移，所以安全检查只涉及到界限检查。处理器保证JMP， CALL， RET 指令不会超出当前执行的代码段的限长。这个限长被存储在CS段寄存器的不可见部分。所以这种检查不会带来额外的时钟周期。

而JMP， CALL， RET 的“FAR”（远）形式则会转移到不同的段内，所以，处理器将执行特权级检查。JMP和CALL有两种方法转移到另一个段：

1、 操作数选择一个另一个可执行段的描述符。

2、操作数选择了一个调用门描述符。这种门形式的转移将在下一节介绍调用门时讲述。

图6-4显示了，两种不同特权级之间的控制转移（没有使用调用门时）：

1、 CPL（当前特权级（Current privilege level））。

2、 目标段的描述符的DPL。

一般来说，CPL和处理器正在执行的段的DPL是相同的。但是，当正在执行的段的一致性位（conforming bit）置位时，CPL也可能比DPL要大。处理器把当前特权级（CPL）缓存在CS段寄存器里，这个值也可能和当前代码段的描述符特权级不同的。

只有当以下条件致少满足一个时，处理器才允许JMP或CALL直接转移到另一个段：

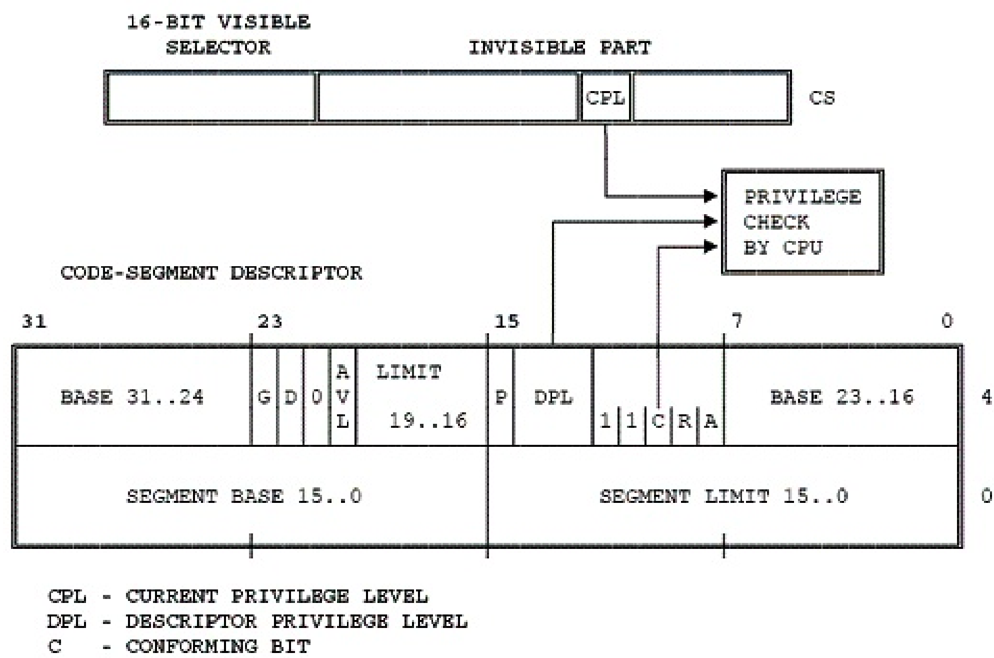
- 目标段的DPL和CPL相同时。
- 目标代码段的一致性位（conforming bit）设置时，而且目标代码段的DPL与CPL相等或者目标代码段的DPL比CPL小。

一致性位设置的段被称为一致性段。一致性段的机制允许不同特权级共享子程序，而且在执行其中的子程序时使用自已的特权级，而不是使用一致性段的段描述符特权级。一个例子就是数学库子程序和一些异常处理子程序。当控制转称到一致性段时，CPL不会改变。这就是唯一的CPL不等于当前可执行代码段的情况。

许多代码都是非一致性的（non-conforming）。上述的基本特权级检查意思是，对于非一致性段，不通过门描述符可以转移到一个相同特权级的可执行段。但是，有时我们也需要从低特权级向高特权级（数值上比较小的）转移。这种需要就可以能过调用门（call-gate）来实

现。调用门在下一节中介绍。JMP 指令不可能通过任何方法转移到DPL与CPL不同的非一致性段中去。

Figure 6-4. Privilege Check for Control Transfer without Gate



6.3.4 门描述符保证子程序入口点（Gate Descriptors Guard Procedure Entry Points）

为了提供转移到不同特权级段中的机制，80386使用了门描述符。有4种门描述符：

- 调用门（Call Gates）
- 陷阱门（Trap Gates）
- 中断门（Interrupt Gates）
- 任务门（Task Gates）

这一章只讲述调用门。任务门是用来任务切换的，所以在第7章中介绍。第9章说明了陷阱门和中断门，用来处理异常和中断。图6-5显示了调用门的格式。一个调用门可以在GDT中，也可以在LDT中，但不能在IDT中。

一个调用门主要有2个作用：

1、 定义一个子程序的入口。

2、 定交了入口的特权级。

调用门描述符被CALL和JMP指令使用，方法和普通的代码段描述符相同。当硬件识别了目标选择子是指向一个调用门时，操作过程将由这个调用门来决定。

门中的选择子和偏移量将用来形成一个子程序入口点的指针。调用门保证了控制转移到一个段内的合法入口点，而不是转移到一个子程序的中间，或者更糟糕的是转移到一条指令的中间。作为操作数的远指针不再象平常那样指向一个段中的某个偏移了，而是选择子部分指向了一个门，偏移部分没有使用。图6-6显示了这种寻址方式。

如图6-7所示，4种不同的特权级将用来做安全检测：

1、 CPL

2、 用来指向调用门的选择子的RPL

3、 门描述符的DPL

4、 可执行目标段的DPL

调用门描述符的DPL决定了什么样的特权级可以使用这个门。一个段可能有多个子程序，这些子程序被设计给不同特权级程序来使用。例如，操作系统可能有几个子程序，这些服务被设计来给应用程序使用，但其它的子程序可能只被设计成给系统软件使用。

门描述符可以用来向高特权级（数值上更小的）或同特权级控制转移（这样实际上没有必要）。只有CALL指令才能用门描述符向高特权级转移。JMP只能使用门描述符向同级特权级转移或向一个一致性段转移。

对于JMP 指令，向一个非一致性代码段转移时，以下两点必须要同时满足，否则处理器引发异常：

MAX（CPL，RPL） <= Gate DPL

Target Segment DPL = CPL

对于CALL指令（或者对于向一致性代码段转移的JMP）以下两点必须要同时满足，否则处理器引发异常：

```
MAX（CPL， RPL） &lt;= Gate DPL
Target Segment DPL &lt;= CPL
```

Figure 6-5. Format of 80386 Call Gate

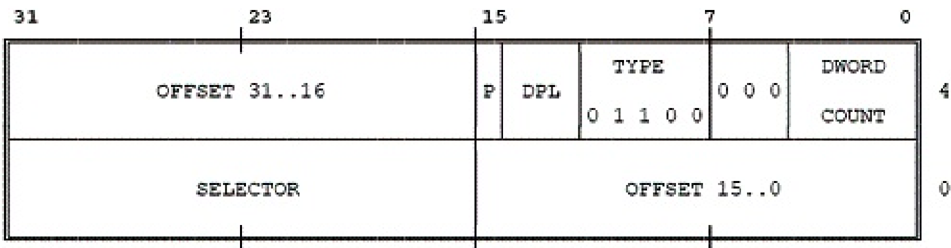
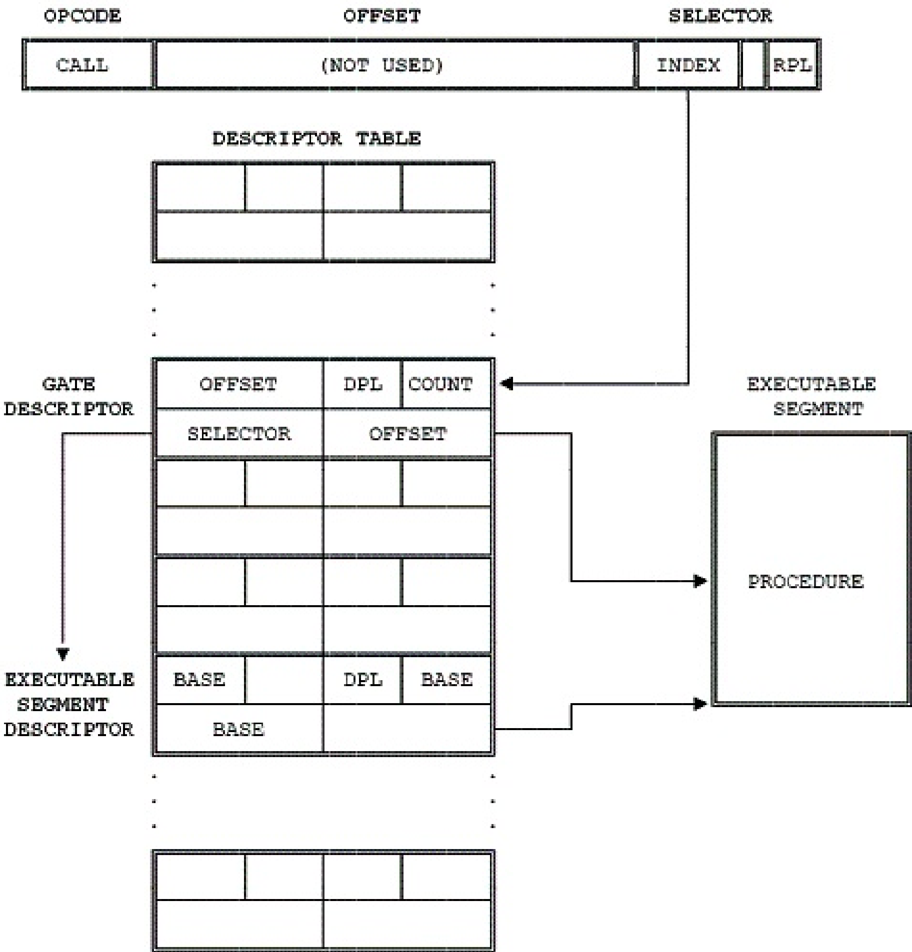
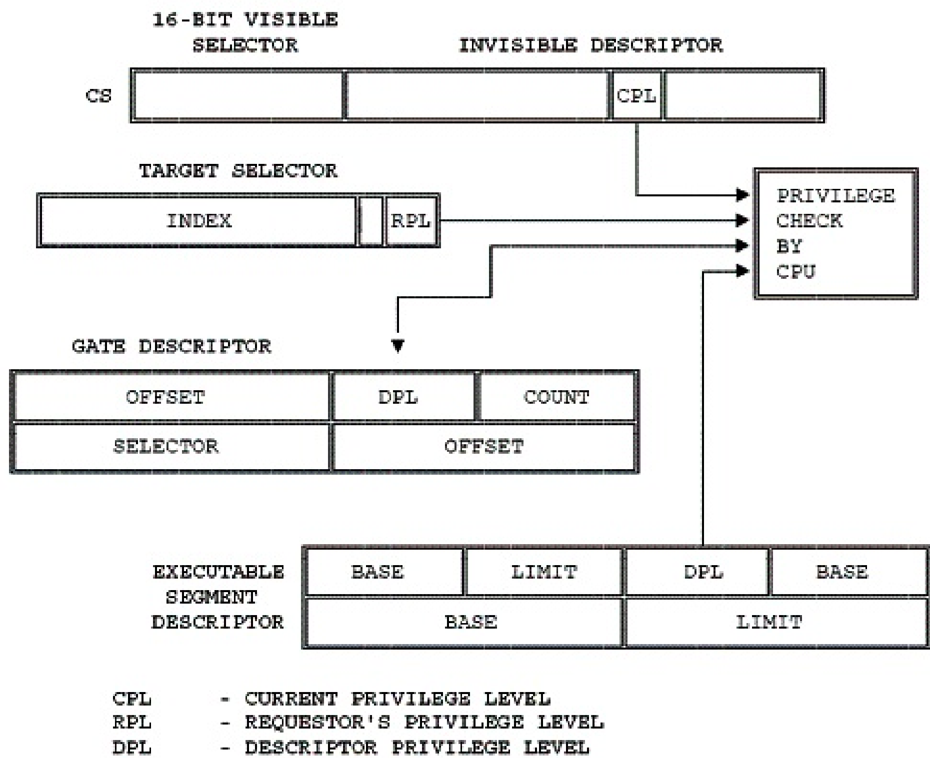


Figure 6-6. Indirect Transfer via Call Gate





6.3.4.1 堆栈切换（Stack Switching）

如果调用门中指示的目标代码段和当前特权级（CPL）不同，段间跳转发生了。

为了保证系统的完整性，每一个特权级使用了一个相互独立的堆栈。这样，可以保证高特权级有足够的堆栈空间使用。没有它们时，如果调用者不提供足够的堆栈空间，那么一个受信任的子程序就无法正常工作。处理器通过任务状态段（task state segment）（请看图6-8）来寻址这些堆栈。每一个任务有一个单独的TSS， 所以允许任务拥有自已的堆栈。系统软件的责任是创建TSS还要把它们的堆栈指针设置好。TSS最初的堆栈指针是只读的。处理器绝对不会在执行程序时更改它们。

当一个调用门用来改变特权级时，处理器使用TSS中的堆栈指针来建立一个新的堆栈。处理器用目标代码段的DPL来索引TSS中的堆栈指针，PL0，PL1 或PL2。

新堆栈的DPL必须和新的CPL相等，如果不是，处理器引发堆栈异常。为每一个特权级创建堆栈和堆栈段描述符是系统软件的责任。每一个堆栈必须包含必要的空间来容纳旧的SS:ESP，返回地址（CS：EIP）和所有的参数，局部变量等。

内部调用时，传给子过程序参数放在堆栈上。为了使特权转移相对被调用者来说透明，处理器把参数拷贝到新堆栈上。调用门的 count 字段说明了有多少个双字参数需要从调用者堆栈上拷贝到新的堆栈上。如果 count 字段为0，则不用拷贝任何参数。

在特权级转移调用过程中，处理器执行以下堆栈相关的操作：

1、 处理器检测新的堆栈是否有足够的空间容纳各参数和返回链。如果不能，则引发一个错误码为0 的堆栈错误异常。

2、 旧的堆栈寄存器 SS：ESP 各以双字的形式压入新的堆栈中。

3、 拷贝参数。

4、 一个在CALL指令后的指令指针（旧的CS：EIP）被压入新堆栈。最后SS：ESP指针将指向新堆栈中的这个返回值。

图6-9显示了在成功调用后的堆栈内容。

TSS 段没有特权级3的堆栈指针保存区，因为特权级3不能被任何一个别的特权级

调用。

被别的特权级调用的子程序，如果需要比31个双字还要多的参数的话，就必须使用保存的SS：ESP链来访问第31个以后的参数了。

通过调用门的子程序调用并不检测拷贝到新栈的参数的值。被调用程序有责任对参数的有效性检测。后面的小节将讨论如何使用 ARPL, VERR, VERW, LSL, 和LAR 指令来检测指针的有效性。

Figure 6-8. Initial Stack Pointers of TSS

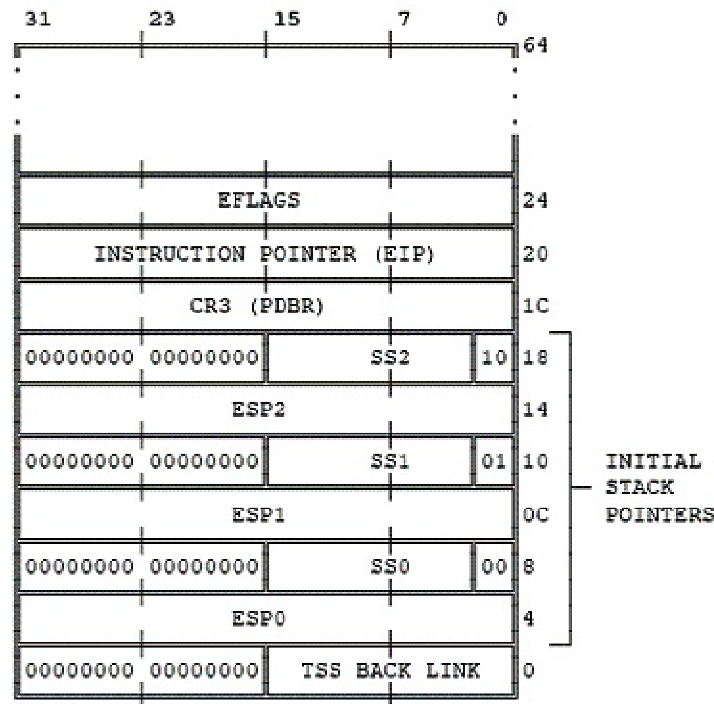
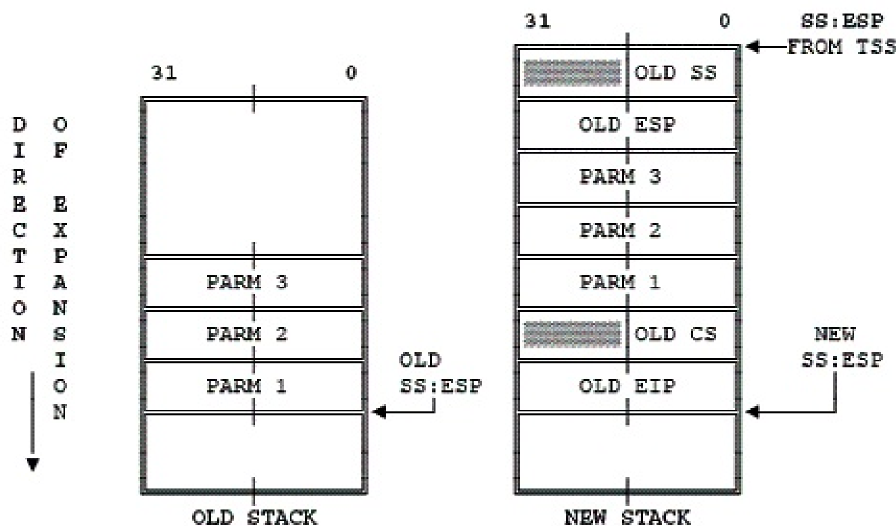


Figure 6-9. Stack Contents after an Interlevel Call



6.3.4.2 从子过程中返回（Returning from a Procedure）

NEAR 形式的 RET 指令只是在当前段内做控制转移，所以只做界限检测。跟在CALL指令后的OFFSET 部分将从堆栈中弹出。处理器保证OFFSET 部分不会超过当前可执行段的界限。

FAR 形式的RET指令把先前通过FAR CALL 指令而压入栈的返回指针从栈中弹出。一般情况下，该值应该是有效的，因为它和先前的CALL和INT对应。但是，处理器还是会执行特权检测，以防止当前执行的子程更改这个指针或者子程序没有对堆栈做正确的维护。 从堆栈中弹出的CS寄存器中的RPL字段指示了调用者的特权级。

段间返回指令可能会改变特权级，但是只能向更低的特权级返回。当一条RET指令遇到一个保存的CS寄存器，而且它的RPL字段比CPL（当前特权级）要数值上更大时，段间返回便发生了。那样的返回执行以下的步骤：

1、 图6-3显示的检测被执行，用以前存储在堆栈中的旧值加载CS：EIP和SS：ESP寄存器。

2、 RET 指令指示要对旧的SS：ESP所做的调整。结果的ESP将不会和堆栈段的界限做检查。如果ESP超出了界限，直到下一次的堆栈操作才会被识别到。（做返回动作的子过程的SS：ESP部分是不会被保存的，一般说来，这个值和TSS中保存的相同）

3、 DS， ES， FS，GS 寄存器的内容将被检查。如果有任何一个寄存器指向了一个比当前特权级高的段（当然除了一致性段），该寄存器将被加载一个NULL选择子（INDEX=0，TI=0）。RET指令本身并不会产生任保异常。任何使用空选择子的以后的内存访问将引发一个通用保护异常（general protection exception）。这样可以防止低特权级程序通过使用高特权级留在堆栈里的选择子来访问高特权级的段。

6.3.5 一些指令是为操作系统保留的（Some Instructions are Reserved for Operationg System）

一些会对保护机制产生影响的指令和会对系统性能产生影响的指令只能由受信任的

代码执行。80386 有两种这样的指令：

1、 特权指令——这些指令用于系统控制

2、 敏感指令——这些是用于I/O或和I/O相关的指令。

Table 6-3. Interlevel Return Checks

Type of Check
SF Stack Fault
GP General Protection Exception
NP Segment-Not-Present Exception

Type of Check	Exception	Error Code
ESP is within current SS segment	SF	0
ESP + 7 is within current SS segment	SF	0
RPL of return CS is greater than CPL	GP	Return CS
Return CS selector is not null	GP	Return CS
Return CS segment is within descriptor table limit	GP	Return CS
Return CS descriptor is a code segment	GP	Return CS
Return CS segment is present	NP	Return CS
DPL of return nonconforming code segment = RPL of return CS, or DPL of return conforming code segment ≤ RPL of return CS	GP	Return CS
ESP + N + 15 is within SS segment N Immediate Operand of RET N Instruction	SF	Return SS
SS selector at ESP + N + 12 is not null	GP	Return SS
SS selector at ESP + N + 12 is within descriptor table limit	GP	Return SS
SS descriptor is writable data segment	GP	Return SS
SS segment is present	SF	Return SS
Saved SS segment DPL = RPL of saved CS	GP	Return SS
Saved SS selector RPL = Saved SS segment DPL	GP	Return SS

6.3.5.1 特权指令（Privileged Instructions）

影响系统数据结构的指令只能在特权级0下执行。如果处理器在当前特权级大于0的情况下遇到这样的指令，将产生一个通用保护异常。这些指令包括：

```
CLTS                — Clear Task-Switched Flag
HLT                 — Halt Processor
LGDT                — Load GDL Register
LIDT                — Load IDT Register
LLDT                — Load LDT Register
LMSW                — Load Machine Status Word
LTR                 — Load Task Register
MOV to/from CRn     — Move to Control Register n
MOV to /from DRn    — Move to Debug Register n
MOV to/from TRn     — Move to Test Register n
```

6.3.5.2 敏感指令（Sensitive Instructions）

当当前特权级不是0时，和I/O相关的指令需要一定的约束条件才能执行。I/O执行机制将在第8章（输入、输出）讲述。

6.3.6 指针有效性检测（Instruction for Pointer Validation）

指针有效性检测是检测程序错误的一个重要手段。指针检测还是保持不同特权级间的独立性的必需。指针检测包括以下几个步骤：

1、 检查指针的提供者有权访问段。

2、 检测段的类型是否可以以指针指示的方式使用。

3、 检查指针没有越界。

虽然80386在指令执行时会自动执行2和3检测，但软件必须要协助来做第1类检测。 非特权指令的作用就是体现在这方面的。软件也可以自已做2和3类检测，以避免处理器产生异常。非特权指令 LAR, LSL, VERR, VERW 就是用于执行这样的操作的。

LAR（Load Access Rights）用来检测一个特权级的指针针访问了合适特权级和正确类型的段。LAR 有一个操作数——想检察属性的段描述符的选择子。描述符必须要可被访问（CPL和选择子的RPL）。如果描述符可被访问，LAR将得到描述符的第二个双字部分，用值0xFxFF00H来掩它，存储到一个32位的目的地寄存器里，设置ZERO 标志（X指的是存储的这4位没有定义）。一旦加载了，便可以对这些访问特权进行测试。

如果RPL或CPL比DPL要大，或者选择子超出了描述符表的界限，则没有访问权限被返回，ZERO位置0。一致性段可以被任意特权级访问。

LSL（Load Segment Limit）允许软件测试一个描述符。如果在当前特权级下可访问该描述符的话，LSL加载32位的。字节计数的、 从界限片段字段计算出来的没有整合的界限，G-位到指定的32位寄存器。只有数据段，代码段，任务状态段和局部描述符表才可以做这些操作；门描述符是不可访问的（表6-4详细列出了哪种类型的可以，哪种类型的不行）。怎么解析界限和段的类型相关，比如，向下增长的数据段对于界限的解析和代码段对界限的解析是不同的。对于LAR和LSL，如果操作执行了，ZERO位被置位，否则ZF位清除。

Table 6-4. Valid Descriptor Types for LSL

Type Code	Descriptor Type	Valid?
0	(invalid)	NO
1	Available 286 TSS	YES
2	LDT	YES
3	Busy 286 TSS	YES
4	286 Call Gate	NO
5	Task Gate	NO
6	286 Trap Gate	NO
7	286 Interrupt Gate	NO
8	(invalid)	NO
9	Available 386 TSS	YES
A	(invalid)	NO
B	Busy 386 TSS	YES
C	386 Call Gate	NO
D	(invalid)	NO
E	386 Trap Gate	NO
F	386 Interrupt Gate	NO

6.3.6.1 描述符的有效性（Descriptor Validation）

80386 有两条指令，VERR 和VERW， 用来测试当前特权级是否有权对一个段的读写。当不可访问时，两条指令都不会产生异常。

VERR（Verify for Reading）检查一个段的可读性，当在当前特权级可读时把ZF置1。VERR做以下检测：

1、 指向描述符的选择子在LDT或GDT的界限之内。

2、 它指向一个代码或数据段描述符。

3、 在一合适的特权级下段可读

对于数据段和非一致性代码段的检查是，DPL必须要同时在数值上大于或等于CPL

选择子RPL。一致性段不做特权检查。

VERW（Verify for Writing）和VERR一样做相似的检测，不过是对于写。和VERR指令一样，当在当前特权级下可写时，VERW置ZF位为1。指令还会检查描述符在描述符表界限内，是一个段描述符，可写，DPL在数值上同时比CPL和选择子的RPL大或相等。代码段永远不可写，不管是一致性还是非一致性的。

6.3.6.2 指针完不整性和RPL

请求特权级（RPL）特性可以防止一个低特权级的不正确的使用指针而破坏高特权级的数据或代码一个很常见的例子是，文件系统子程序，FREAD（file_id, n_bytes, buffer_ptr）。这个假设的子程序从一个文件读取数据，然后把数据放入缓冲区，不管缓冲区内是任何数据，都将被覆盖。一般情况下，FREAD对于用户程序是可见的。在没有指针检测的标准下，一个用户程序可能提供一个文件表的指针而非一个缓冲区的指针，以至使FREAD子程序使文件表受到损坏。

使用RPL可以避免这种问题。RPL字段允许赋一个特权级给选择子。这个特权级属性一般指出了产生选择子的代码的特权级。当选择子被加载时，80386会自动检查RPL是否允许访问。

为了更好地利用好处理器对RPL的检测，被调用的子过程只用确保传给它的选择子至少在数值上比CPL大就行了。这样就可以使选择子被赋予比它们的提供者更低的特权级。如果一个选择子用来访问一个调用者都不能访问的段时，也就是说RPL在数值上比DPL更大，当加载该选择子时便会产生保护异常。

ARPL（Adjust Requestor’s Privilege Level）调整一个选择子的RPL字段或一个寄存器的RPL字段，使RPL变大。后者一般都是从一个保存在堆栈上的CS寄存器的映象加载的。如果请求特权级被调整，ZF位置1，否则置0。

6.4 页级保护（Page-Level Protection）

两种保护机制和页级保护相关：

1、 可寻址范围约束。

2、 类型检查。

6.4.1 页表项保存保护参数（Page-Table Entries Hold Protection Parameters）

图6-10高亮显示了控制访问的页表项和页目录项的字段。

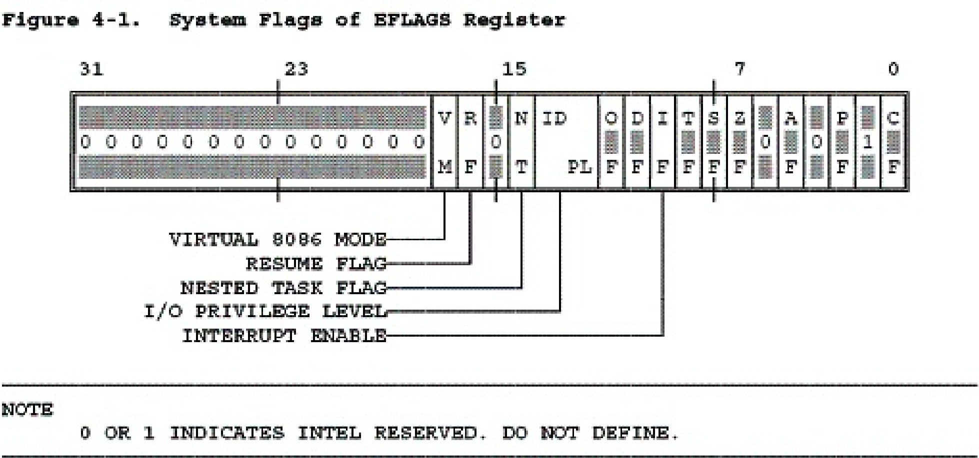


Figure 4-1. System Flags of EFLAGS Register

6.4.1.1 可寻址范围约束（Restricting Addressable Domain）

页面的特权级的概念是通过以下两级来实现的：

1、 超级用户级（Supervisor level （U/S=0））—— 用于关连操作系统和其它一些系统软件和数据。

2、 用户级（U/S=1）—— 用于应用程序子程序和数据。

当前级（U或者S）和CPL相关的。如果CPL是0，1或2，系统在特权级执行。如果CPL是3，处理器在用户级执行。

当系统在超级用户（特权级）模式执行，所有页面可寻址，但是，当处理器在用户模执行时，只有用户的页面可寻址。

6.4.1.2 类型检查（Type Checking）

在分页地址转换时，以下两种类型被定义：

1、 只读访问（R/W=0）（Read-Only Access）

2、 可读写访问（R/w=1）（Read/Write Access）

当处理器在特权模式下执行时，所有页面都是可读可写的。哪处理器在用户模式下执行时，只有用户页面而且被标识为可写的页面才能写，被标识为只读的页面则只允许读取。所有属于超级用户的页面都不可访问，无论读还是写。

6.4.2 混合两级页表保护

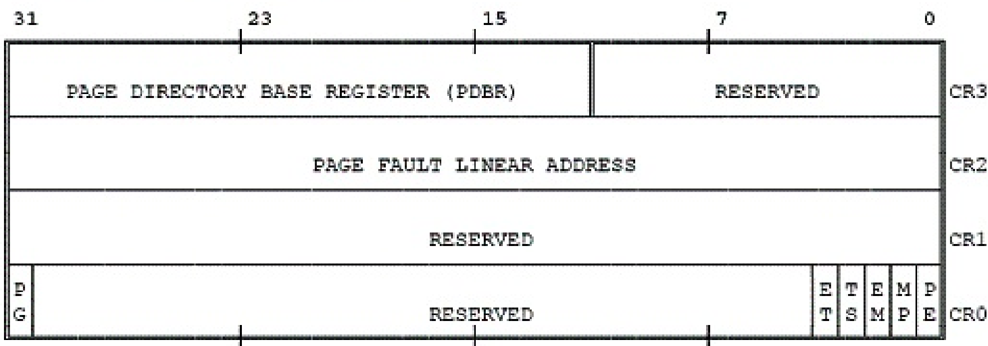
对于任何一个页面，它的页目录项可能和页表项的保护属性不同。80386综合计算两级页表的保护参数来保护一个页面。表6-5显示了这种保护。

6.4.3 覆盖页保护（Overrides to Page Protection）

一些访问会使用特权级0来当做访问发出者做检测，即使CPL=3时。

1、对LDT，GDT、TSS、IDT的访问。

3、 通过跨特权级的CALL/INT来访问内层堆栈。



6.5 混合分页和分段保护（Combining Page and Segment Protection）

当启用分页时，80386先执行分段保护检测，再执行分页保护检测。当任一级发生了违反保护机制时，处理器产生一个保护异常。

例如，可以定义一个很大的段，有一些单元只读，而其它的单元则可读写。在这种情况下，页目录（或页表）项，可以把想要设成只读的单元的表项的 U/S 和 R/W 位置0，指示为所有该项指示的页面都不可写。这种技术可以用于类UNIX系统上，来定义一个大的数据段，部分只读（共享数据和ROM 常量）。这样就允许类UNIX系统定义一个大的段为一个“平坦”的数据空间，用“平坦”的指针来寻址这个“平坦”的空间，也同样能完成只读数据的保护，共享文件映射到虚拟内存空间，和特权用户区域。

第7章 多任务（Multitasking）

为了更好的保护好多个任务，80386使用了几种特别的数据结构。但是，并没有使用特别的指令来控制多任务。相反，当遇到转移指令是访问的特别的数据结构时，它用不同的方法来解析控制转移。用来控制多任务的寄存器和数据结构是：

1、 任务状态段（Task state segment）

2、 任务状态段描述符（Task state segment descriptor）

3、 任务寄存器（Task register）

4、 任务门描述符（Task gate descriptor）

有了这些数据结构，80386可以快速的从一个任务切换到另一个任务中去，把原先任务的上下文（context）保存起来，以便以后可以重起该任务。除了任务切换以外，80386还进行以下两个任务管理：

1、 中断和异常可以引起任务切换（如果系统设计需要的话）。处理器不但切换到中断处理程序的任务中，而且当中断处理完后还会自动返回原任务。中断任务可以中断低特权级的任务，无论多少级。

2、 当每一次切换到另一个任务时，80386也会切换到另一个LDT和另一个页目录去。这样，每个任务都有了不同的逻辑地址——线性地址，和线性地址——物理地址的映射了。这是另一个保护的特性，它把任务独立开来，以防止它们之间的相互干涉。

8.1 I/O 寻址（I/O Addressing）

80386 允许以以下的两种方式操作输入、输出：

- 通过独立的I/O地址空间（使用特定的I/O指令）
- 通过内存映射I/O（使用一般的指令操作数）

7.1 任务状态段（Task State Segment）

用来管理任务的所有信息都被保存在一个特别的段中，任务状态段（TSS）。图7-1 显示了80386的TSS的格式（另一种类型用来执行80286任务，参看第13章）

TSS 状态段由两部分组成：

1、 动态部分，处理器在每次任务切换时会设置这些字段值：

- 通用寄存器（EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI）。
- 段寄存器（ES，CS，SS，DS，FS，GS）
- 状态寄存器（EFLAGS）
- 指令指针（EIP）
- 前一个执行的任务的TSS段的选择子（只有当要返回时才更新）。

2、 静态字段，处理器读取，但从不更改。这些字段包括：

- 任务的LDT选择子
- 页目录基址寄存器（PDBR）（当启用分页时，只读）
- 内层堆栈指针，特权级0-2
- T-位，指示了处理器在任务切换时是否引发一个调试异常。（关于调试信息，参看第12章）
- I/O 位图基址（关于I/O位图的信息，请参看第8章）

任务状态段可以位于线性地址的任意处。唯一一个要注意的问题是，当TSS跨了一个页的边界时，而且第二个页面又不存在时。在这种情况下，当处理器在任务切换时，读一个TSS如果发现页不存在的话，则引发一个异常。这样的异常可以通过以下两种方法来避免：

1、 把TSS只分配到一个页面中，以使它不跨越页边界。

2、 在任务切换时，保证要么两个页都在内存中，要么都不在内存中。如果两个页面都不在内存中的话，缺页中断处理程序必须在重起用于任务切换的指令前把两页都读入内存中。

Figure 4-1. System Flags of EFLAGS Register

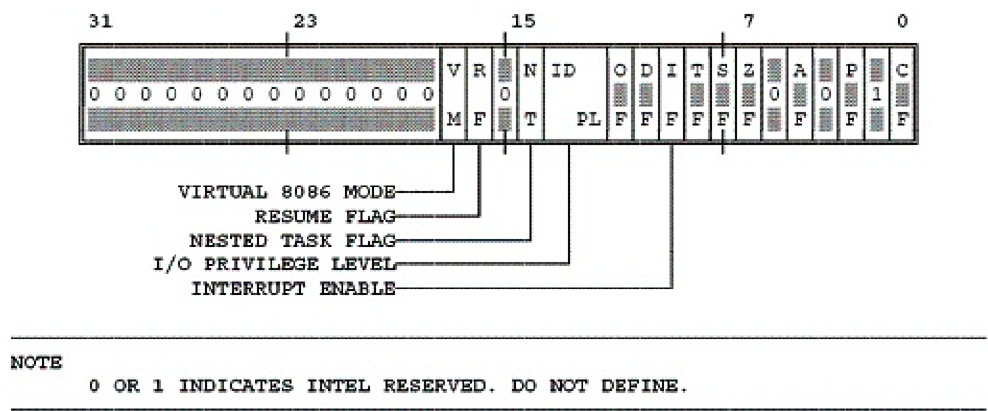
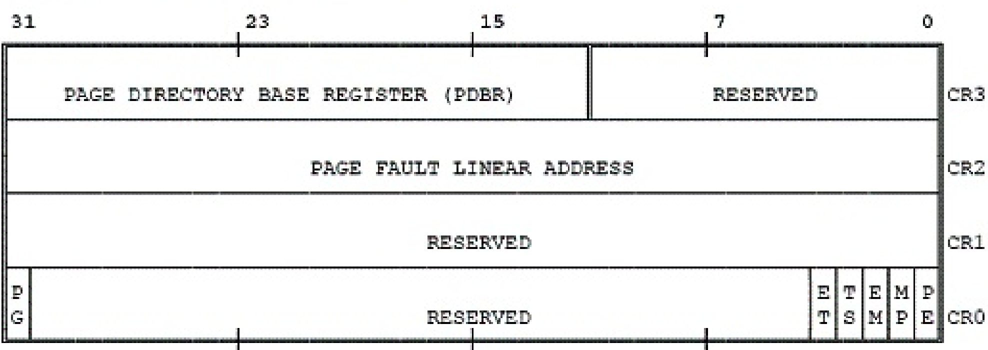


Figure 4-2. Control Registers



7.2 TSS 描述符（TSS Descriptor）

和其它段一样，任务状态段也是用一个描述符来定义的。图7-2显示了TSS描述符的格式。

类型字段中的B-位指出任务是否忙。类型字段为9指出了一个不忙的任务。类型字段为11指出了忙的任务。任务是不可重入的。B-位可以使处理器检测到一个向忙的任务切换的操作。

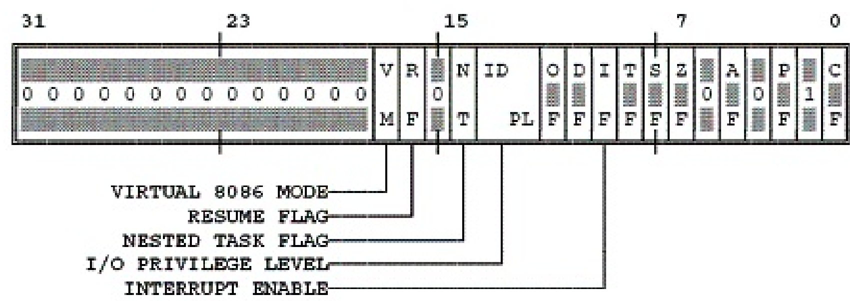
BASE，LIMIT，DPL，字段和G-位、还有P-位，与其它数据段描述符的作用类似。LIMIT字段，必须大于或等于103。如果在任务切换中，发现任务描述符的界限字段小于103字节的话，处理器引发异常。更大的界限是允许的，如果I/O许可位图存在的话，更大的界限就昌必要的了。如果系统软件想要在TSS中存放额外的数据的话，更大的界限也是可能的。

一个能访问TSS描述符的子程序可以引起任务切换。在大多数系统中，TSS的DPL字段应该设置为0，所以只有最受信任的软件才能做任务切换。

有访问一个TSS的特权，并不意味着可以读或更改TSS。读和更新一个TSS段必须用另一个描述符来重定义成一个数据段。任何想要把TSS描述符加载到一个段寄存器（CS，SS，DS，ES，FS，GS）将产生异常。

TSS描述符只能位于GDT中。如果用一个选择子而且TI=1（指示在当前的LDT内）来标识一个TSS的话将产生一个异常。

Figure 4-1. System Flags of EFLAGS Register



NOTE

0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE.

7.3 任务寄存器（Task Register）

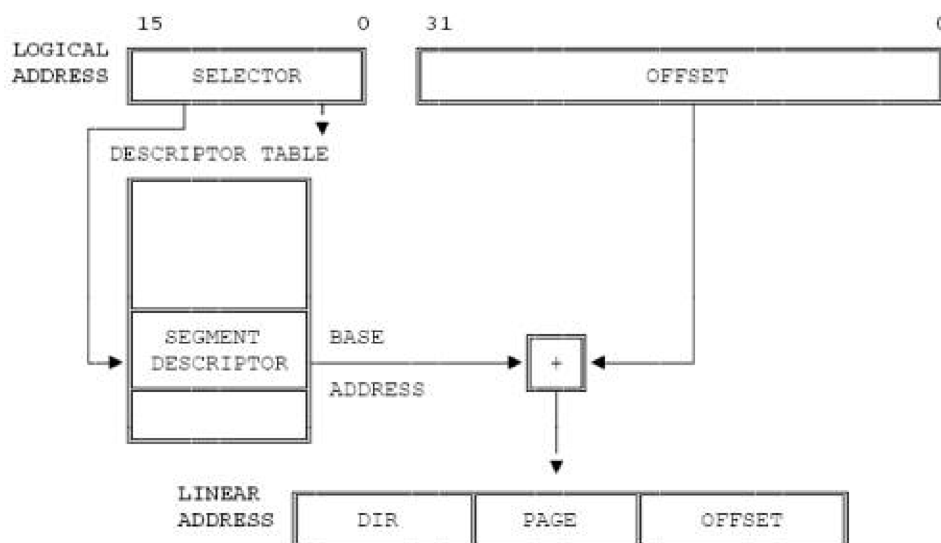
任务寄存器（TR）通过指向一个TSS，寻址了当前正在执行的任务。图7-3显示了处理器如何访问当前任务的TSS。

任务寄存器有一个“可见部分”（也就是说，可以被指令读写的部分）和一个“不可见部分”（由处理器操作，对应着可见部分，不可以通过指令来读写）。可见部分的选择子部分选择了一个在GDT中的TSS。处理器用不可见部分来缓存TSS描述符中的基址和界限值。把基址和界限保存在一个寄存器中可以提高任务的执行性能，因为处理器不必每次都访问内存来得到当前任务TSS的这些值。

LTR指令和STR指令是用来更改和读取任务寄存器的可见部分的。两条指令都有一个操作数，一个在内存中的或在通用寄存器中的16-位选择子。

LTR（Load task register）加载一个选择子操作数到任务寄存器的可见部分，这个选择子必须指定一个在GDT中的TSS描述符。LTR也用TSS中的信息来加载任务寄存器的不可见部分。LTR是一条特权指令，只能当CPL是0时才能执行这条执令。LTR一般是当操作系统初始化过程执行的，用来初始化任务寄存器。以后，任务寄存器（TR）的内容由每次任务切换来改变。

STR（Store task register）存储任务寄存器的可见部分到一个通用寄存器或者到一个内存的字内。STR不是特权指令。



7.4 任务门描述符（Task Gate Descriptor）

一个任务门描述符提供了一个间接的、有保护性的对一个TSS的的访问方法。图7-4显示了任务门的格式。

门描述符的选择子（SELECTOR）字段必须要指向一个TSS描述符。在这个选择子内的RPL字段是不被处理器使用的。

门描述符的DPL字段用于控制可以访问该描述符来导致任务切换的特权级。只有当选择子的RPL和子程序的CPL的最大值在数值上小于或等于描述符的DPL，这个特性防止了非受信任代码引起任务切换（注意，当使用任务门时，目标TSS描述符的DPL字段不用来做特权级检测。）

和一个可以访问一个TSS描述符的子程序一样，一个有权访问门描述符的子程序就可以引起任务切换。80386使用门描述符来达到以下三个需要：

1、 使一个任务只有一个忙位。因为忙位（busy-bit）存储在TSS描述符中，每一个任务吸能有一个这样的描述符。也可能有这样的情况，几个任务同时选中同一个TSS描述符。

2、 提供可选的其它方式来访问任务。任务门可以满足这种要求，因为他们可以存放在LDT中，还可以和要访问的TSS有一个不同的DPL字段。一个不能访问GDT中的TSS的程序，也可以通过任务门来通过自已的LDT访问该任务。有了门描述符，系统软件可以把任务切换只限制一定的权限下。

3、 为了让中断和异常可以引发任务切换。任务门可以存放在IDT中，从而允许中断或异常引起任务切换。当IDT中的项包含一个门描述符时，80386处理器切换到指定的任务。以便系统中的所有任务和中断任务分离开来。

图7-5显示了在LDT和IDT中指向同一个任务的门描述符。

Figure 5-2. Segment Translation

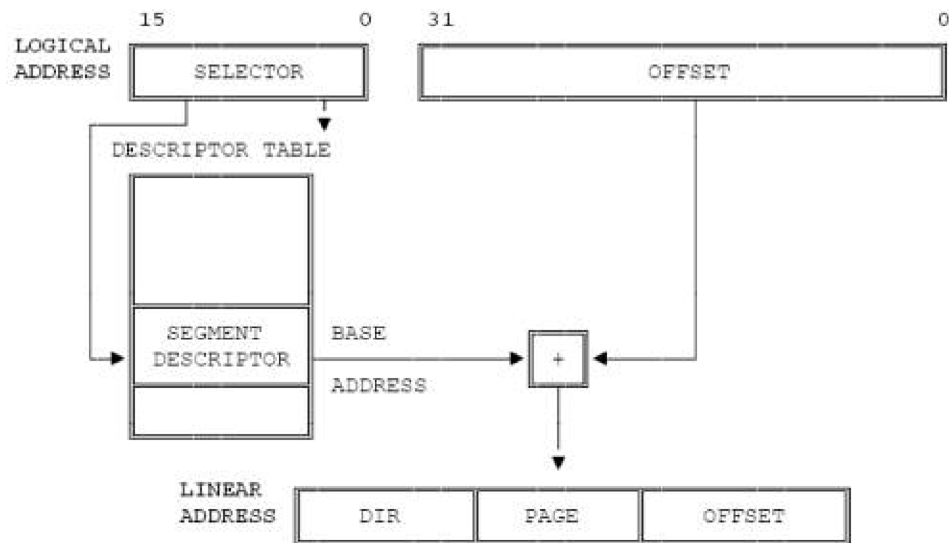
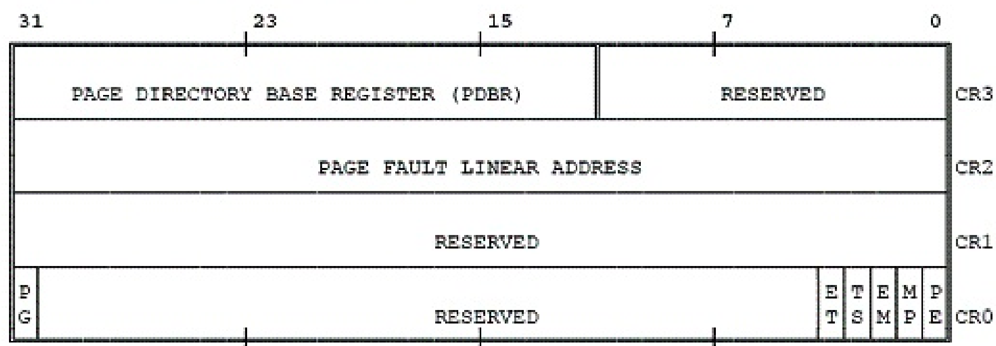


Figure 4-2. Control Registers



7.5 任务切换（Task Switching）

以下情况中，80386切换到另一个任务执行：

1、 当前任务执行了一个JMP或CALL，而操作数中指定了一个TSS描述符。

2、 当前任务执行了一个JMP或CALL，而操作数中指定了一个任务门。

3、 一个在IDT中的中断向量或异常向量导致向一个新的任务切换。

4、 当前任务执行了一条IRET指令，而且NT位设置时。

JMP, CALL, IRET, 中断和异常原先被设计用于在同一个任务内的机制，不需要任务切换。访问到何种类型的描述符还是在标志字段NT（nested taks）位可以用于区分出标标准的机制还是变种的任务切换机制。

为了引起任务切换，JMP或CALL指令可以指定一个TSS描述符或者一个任务门。两种情况下作用是相同的：80386切换到指定的任务。

当在IDT中的中断或异常向量指示了一个任务门时，中断或异常将引起任务切换。如果指示了一个IDT中的中断门或陷阱门，不发生任务切换。关于中断的更多信息，请参看第9章。

当以一个任务或一个中断子程序来引发时，中断处理程序总是将控制返回到被中断任务的子程序。如果NT位被置位，中断处理程序则是一个中断任务，IRET指令将返回到被中断的子程序。

任务切换操作将做以下的步骤：

1、 检测当前任务有权切换到指定的任务。这时数据访问规则将用于检测JMP或CALL指令。TSS描述符或者任务门的DPL字段必须小于或者等于CPL和门选择子RPL字段的最大值。中断、异常、IRET指令可以切换到任何任务，而不必管目标TSS描述符或者目标任务门的DPL字段。

2、 检测目标TSS描述符存在的，而且有一个有效的界限值。到这时，所有的错误都算是在的引发任务切换（outgoing task）的上下文中发生的。错误是可以被处理和重起的，且对于应用程序是透明的。

3、 保存当前任务的状态。处理器从任务寄存器中缓存的不可见部分来找到当前任务的基址。处理器拷贝寄存器值到当前任务TSS（EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, 和 标志寄存器EFLAG）。EIP字段则指向引起任务切换的指令的下一条指令。

4、 将新的任务的选择子加载到任务寄存器，将新任务的TSS描述符设置为忙。设置MSW的TS（task switched）标志位。选择子或是从指令操作数中得到，或是从任务门中得到。

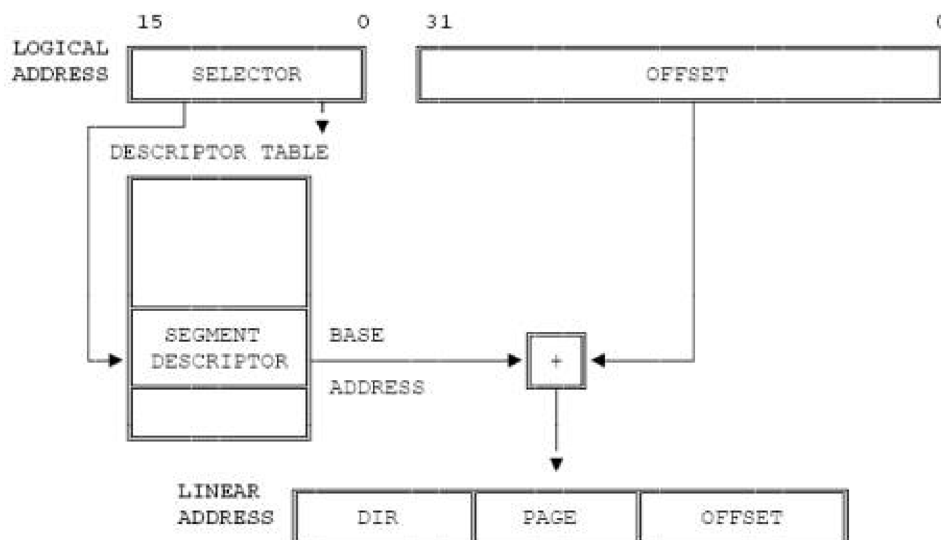
5、 从新的任务的TSS中加载任务的状态，并恢复其执行。加载的寄存器是LDT寄存器，标志寄存器（EFLAG）通用寄存器EIP，EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI；段寄存器ES，CS，SS，DS，FS，和GS。还有PDBR（CR3）。所有检测到的错误将发生在新任务的上下文中。对于一个异常处理程序，看来起好似新任务的第一条指令还未执行。

注意，不管怎么样，旧任务的状态总是会被保存。如果这个任务被重新执行，它执行引起任务切换的指令的后一条指令。当任务执行时，所有寄存器的值将被恢复。

每一次任务切换都会设置MSW（machine status word）的TS（task switched）位。TS标志对于有协处理器的系统来说是很重要的。TS位指出了协处理器的状态可能和当前任务的状态不一致了。第11章进一步讨论TS位。

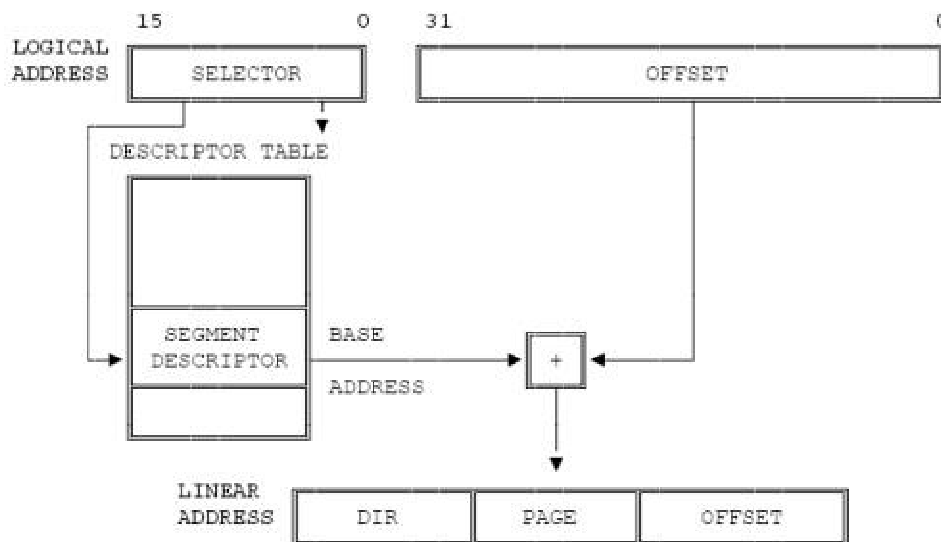
处理任务切换异常的处理程序（表7-1中由第4到16引起的异常）应该注意加载引起异常的选择子的操作。这样的操作可能引发第二次异常，除非异常处理程序首先检查了选择子并修定了潜在的问题。

将要执行的任务的特权级即不被引起任务切换的任务所影响，也不会被它所约束。因为每个任务的地址空间是分开的，且有不同的TSS，还有就是特权级规则可以用于防止不合法的TSS访问，但是没有哪种特权级规则需要用来去约束不同任务间的CPL。新的任务将在CS选择子的RPL字段特权级执行，这个CS是由TSS中加载的。



7.6 任务链（Task Linking）

TSS的返回链（back-link）字段和标志字中的NT（nested task）位允许80386自动返回到一个先前调用任务或被中断的任务中去。当一条CALL指令，或中断指令，或内部中断，或一个异常引起了任务切换，到了一个新任务中。80386处理器自动使用当前任务的选择子来填充新作任务的返回链字段，同时设置新任务标志寄存器的NT位。NT位指示出返回字段是否有效。新的任务通过IRET指令放弃当前控制。当解析IRET指令时，386检查NT标志。如果NT位设置，80386切换到由返回字段指示的任务。表7-2总结这些字段的用处。



7.6.1 忙位防止了环（Busy Bit Prevents Loops）

TSS描述符的B-位（busy bit）保证了返回字段的完整性。返回链可能会很深，这种情况发生在一个中断任又被别的任务中断时，或一个被调用的任务又调用别的任务时。忙位可以让CPU检测到这样发生一个环的情况。环的形成是指试图进入一个已忙的任务。但是TSS是不可重入的。

处理器照以下方法使用忙位：

1、 当任务切换时，处理器自动设置新任务的忙位。

2、 当任务切换时，如果旧任务没有位于返回链上的话（也就是说引起任务切换的是一条JMP或IRET指令）处理器自动清除旧任务的忙位。如果任务位于返回链上，忙位被保留。

3、 当向一个任务切换时，如果处理器发现新任务的忙位设置，则引发一个异常。

通过这些操作，处理器防止了任务切换到自已或是到一个已在任务链上的任务，所以防止了非法的任务重入。

忙位对于多处理器环境也是有效的，因为处理器在清除或设置忙位时会自动锁定总线。这样的操作可以防止2个处理器同时唤醒同一个任务（关于多处理器，请参看第11章）。

7.6.2 修改任务链（Modifying Task Linkages）

所有对返回链和忙位的修改都应是由最受信任的代码来操作的。这样的修改可能会使一个被中断的任务比一个中断它的任务先恢复执行。受信任代码必须在以下两种方针下将一个任务从返回链中移去：

1、 首先修改中断任务的返回链字段，然后清楚被移除的任务的描述符的忙位。

2、 保证在修改返回链和忙位时，没有中断发生。

7.7 任务寻址空间（Task Address Space）

TSS中的LDT选择子和PDBR字段给了软件系统设计者一个可伸缩性的段页式映射特性。通过为每一个任务一定的段页映射选择，任务可以共享地址空间，可以有不同与其它任务的很大的地址空间，或者以两种极端来共享。

每个任务有不同的地址空间的特性是80386的一个重要的保护特性。如果模块间没有共享地址空间的话，在一个任务中的模块不可以干涉另一个任务中的模块。80386的这种可伸缩性的内存管理机制允许系统设计者把要想协作的在不同任务中的模块设计成部分的地址空间共享。

7.7.1 任务线性——物理地址空间映射（Task Linear-to-Physical Space Mapping）

一般来说，安排任务的线性——物理地址空间的映射可以分为以下两类：

1、 一个线性——物理地址空间被所有任务共享。

当分页没有启用时，这是唯一的一种选择。没有启用分页的话，所有线性地址空间映射到相同的物理地址空是上。

当启用分页时，这种线性地址——物理地址的映射方式导致了所有任务使用一个页目录。如果操作系统支持页级虚拟内存的话，这种线性空间可能超过了物理地址空间。

2、 线性——物理地址空间部分重叠映射。

这种实现的话，要为每一个任务使用不同的一张页目录。因为PDBR（page directory base register）在每次任务切换时被加载，每个任务可以有不同的页目录。

在这样的技术里，不同任务的线性空间可能被映射到不同的物理地址空间上。如果页目录表项指向不同的页表，页表项又指向不同的物理页面的话，任务就不共享任何物理地址空间。

在实际中，所有任务的线性空间中的一部分必须要映射到相同的物理地址空间上去。任务状态段必须要位于一个公共的空间上，以便在处理器做任务切换，更新和读取TSS信息的时地址映射都相同。被GDT映射的线性空间也应该被映射到公共的物理地址空间上。否则，GDT的目的未做定义。图7-6显示了以共享页表的形式如何将线性地址空间的两个任务映射到相同的物理地址空间上。

7.7.2 任务逻辑地址空间（Task Logical Address Space）

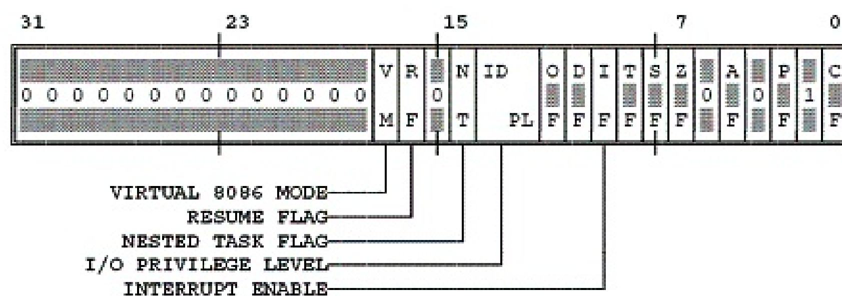
如果只是通过线性——物理地址空间的映射将不能达到任务间数据的共享。为了共享数据，任务必须还要有一个公共的逻辑——线性地址空间的映射。也就是说，他们必须要有访问指向共享线性地址空间的描述符的权限。有3种办法可以建立公共的逻辑地址——物理地址空间的映射：

1、 通过GDT。所有任务都可以通过GDT访问描述符。如果这些描述符指向了一个映射到公共物理空间的线性地址空间的话，所有任务都共享了数据和指令。

2、 通过LDT。如果两个任务的TSS中的LDT选择子部分指向了相同的LDT。这些LDT又包含了一些指向相同物理空间的线性空间的描述符。这种共享的方法比在GDT中共享要更有选择性一点，这种共享可以只局限在几个任务中。其它一些有着不同LDT的任务不可以访问这些公共区域。

3、 通过在LDT中使用别名（aliases）技术。不同的LDT可以包含一些指向相同线性空间的描述符。如果这些线性空间被映射到相同的物理地址空间上去的话，这些描述符就允许任务间共享公共的地址空间。这些描述符被称作“别名”。这种共享机制比前两种更好。在LDT中的其它的描述符也以指向不同的线性空间中。

Figure 4-1. System Flags of EFLAGS Register



NOTE
0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE.

第8章 输入 输出

这一章中从以下几点讲述了80386的I/O特性：

- I/O端口寻址方法（Methods of addressing I/O ports）
- 引起I/O操作的指令（Instructions that cause I/O operations）
- 当使用I/O指令和访问I/O端口时使用的保护规则（Protections as it applies to the use of I/O Instructions and I/O port addresses）

8.1.1 I/O 地址空间 （I/O Address Space）

80386提供了独立的I/O地址空间，不同于物理内存空间，这些空间可以为16位设备端口来寻址。I/O地址空间由此2^16（64K）独立的可寻址8-位端口组成。任何两个连续的8-位端口可以当作一个16-位端口，4个连续的8-位端口可以当作一个32-位的端口。所以I/O地址空间总计64K个8-位端口，或者说32K个16-位端口，或者说16K个32-位端口。

程序可以以两种方式指定端口地址。通过使用立即数，程序可以指定：

- 256个8-位端口，从0~255。
- 128个16-位端口，从0，2，4，……，252，254。
- 64个32-位端口，从0，4，8，……，248，252。

通过使用DX来指定：

- 8-位端口，编号从0到65535
- 16-位端口，编号从0，2，4，……，65532，65534
- 32-位端口，编号从0，4，8，……，65528，65532

80386可以一次传送给指定地址的外设32、16、或8位数据。和在内存中的双字一样，32-位的端口地址应该可以被4整除，似便32-位数据可以在一次总线周期内传送完。16位的应该被2整除，8位的可以指定任何的地址。

IN和OUT指令在寄存器与端口间传送数据。INS和OUTS在内存和I/O地址空间之间传送整串的数据。

8.1.2 内存映射 I/O

I/O设备也可以放在主内存空间中。只要设备象内存一样的作出合适的反应，处理器并区别不出他们。

内存映射I/O指供了外加的编程伸缩性。任何访问内存的指令都可以用来访问位于内存空间的I/O设备。例如，MOV指令可以在寄存器和端口间传送数据。还有AND，OR，和TEST指令可以用来控制在外设内部的寄存器的位（见图8-1）。内存映射I/O可以使用任何寻址模式的指令（真接，间接，其址，索引，标量等）。

内存映射I/O和其它任何内存访问相同，在保护模式下有相同的保护的控制。参看第6章，关于内存保护。

8.2 I/O 指令（I/O Instructions）

80386的I/O指令使得处理器可以访问I/O端口，以便从外设输入数据，或者向外设发送数据。这些指令有一个指定I/O空间端口地址的操作数。有两类的I/O指令：

1、 在寄存器指定的地址传送一个数据（字节、字、双字）。

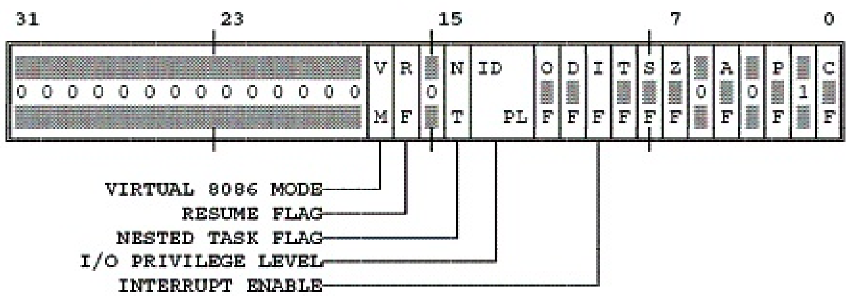
2、 传送指定内存中的一串数据（字节串、字串、双字串）。这些被称作为“串 I/O指令”或者说“块I/O指令”。

8.2.1 寄存器I/O指令（Register I/O Instructions）

I/O指令IN和OUT是用来在I/O端口和EAX（32位）或AX（16位）或AL（8位）通用寄存器间传送数据的。IN和OUT指令可以是直接寻址（0~255端口地址），也可以通过DX寄存器间接寻址（0~64K端口地址）。

IN（Input from Port）从I/O端口传送一个字节、字、双字到AL、AX、或EAX寄存器。如果程序指定了AL寄存器，处理器从端口传送8位到AL寄存器。如果指定了AX寄存器，则传送16位到AX寄存器。如果指定了EAX寄存器，则传送32位到EAX寄存器。

OUT（Output to Port）从AL、AX、或EAX传送一个字节、字、双字到端口。程序可以指定不同的寄存器（AL、AX、EAX）来传送不同数量的字节。



8.2.2 块I/O指令（Block I/O Intructions）

块I/O指令INS和OUTS用来在内存和端口间传送数据。块I/O指令使用DX来指定I/O端口地址。INS和OUTS用DX来指定：

- 8位端口，编号从0~65535
- 16位端口，编号从0，2，4，……，65532，65534

- 32位端口，编号从0，4，8，……，65528，65532

块I/O指令使用SI或DI来指定源或目标内存地址。对每次传送，SI或DI通过标志寄存器里的方向位会自动增加或减少。

INS和OUTS，当使用 repeat 前缀时，会将一块数据输入或输出。REP，重复前缀，修饰INS或OUTS来指示他们在内存和端口间传送一块数据。这些块I/O指令是基于串原语的（参看第3章，关于串原语）。他们使得编程简化了，还能过消除了用单寄存器来保存数数的循环从而增加了数据传输速度。

基于串的指令可以传送字节、字、或是双字串。每一次传送结束后，ESI或EDI的内存地址将更新1个字节（字节操作数），或2个字节（字操作数），或4个字节（双字操作数）。标志寄存器里的DF标志将决定是增加还是减少ESI、EDI（DF=0 增加，DF=1 减少）。

INS（Input String from Port）从一个输入端口传送一串字节、字、或者双字到内存中。INSB、INSW、INSD是这条指令的变种，分别指定了操作传送单元的大小。如果一个程序指定INSB，处理器从输入端口传送8位到ES：EDI指定的内存处。如果程序指定了INSW则传送16位数据单元，INSD则传送32位数据单元。目的地的段寄存器ES不能被更改。和REP前缀一起使用，INS从输入端口传送一块数据信息到连续的内存地址处。

OUTS（Output String to Port）从内存传送一个字节、字、或双字串到输出端口。OUTSB，OUTSW，OUTSD是这条指令的变种，指定的数据单元的大小。如果程序指定了OUTSB，处理器从ES：EDI指定的内存处传8-位到输出端口。如果指定了OUTSW，则传送16位数据单元。如果指定了OUTSD，则传送32位数据单元。混合REP前缀，OUTS从内存的连续地址处传送一块数据到指定的端口。

8.3 保护和I/O（Protection and I/O）

有两种机制用于I/O保护：

1、 FLAGS里的IOPL字段定义了使用I/O指令的特权级。

2、 TSS段里的I/O许可位图定义了使用I/O地址空间的特权级。

这些机制只在保护模式下工作，包含虚拟8086模式。在实模式中他们不会起作用。在实模式中，I/O空间没有任何保护。任何子程序可执行I/O指令。任意I/O端口都是可以被I/O指令寻址的。

8.3.1 I/O 特权级（I/O Privilege Level）

处理I/O的指令应该被限制执行，但不在特权级0执行I/O指令也是需要的。所以，处理器用了标志寄存器里的两位来存储I/O的特权级（IOPL）。IOPL定义了要执行I/O指令所需要的特权级。

以下指令只有当CPL <= IOPL时才可以执行：

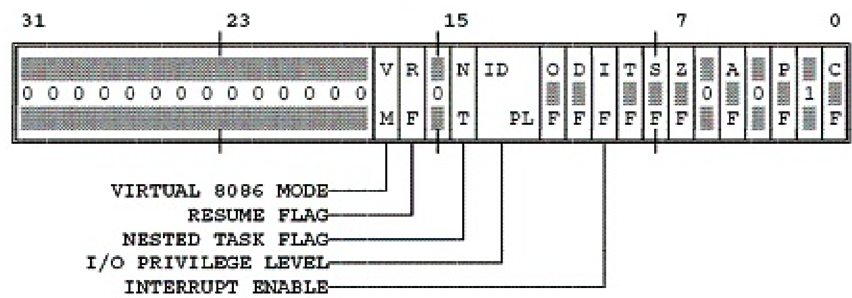


Figure 4-1. System Flags of EFLAGS Register

这些指令被称作“敏感指令”，因为他们对I/O敏感的。

为了执行敏感指令，子程序必须执行在至少小于或等于（数值上）IOPL（CPL <= IOPL）的特权级。任何没有足够特权级的程序执行I/O指令时将引起通用保护异常。

因为每个任务都有自已的标志寄存器，每个程序都有不同的IOPL。一个任务如果主要任务是完成I/O操作的话（设备驱动程序）可以让IOPL为3，所以他的任何子程序都可以执行I/O。其它一些程可以把IOPL设成0或1，只让特权级高的子程序可以执行I/O。

一个任务只有通过POPF指令才能改变IOPL。但是，这样的改变是特权级的。只有在特权级0执行的程序才可以改变IOPL。不够特权级的程如果试图更改IOPL的话，不会产生任何异常，而IOPL只是保持不变。

一条POPF指令可以用来副加的开中断和关中断。但是，通过POPF来改变IF标志位也是特权级的。一个想通过POPF指令来改变IF标志的程序，必须要执行在到少IOPL的特权级上（数值上小于或等于）。同样，不够特权级的试图更改并不会产生任何异常，而IF只是保持不变。

8.3.2 I/O 许可位图（I/O permission Bit Map）

直接指定处理器I/O地址空间的I/O指令是IN，INS，OUT，OUTS。80386可以有选择性的把一些I/O地址空间访问设成陷阱。允计这样做的数据结构是在TSS段（请看图8-2）中的I/O 许可位图（I/O Permission Bit Map）I/O许可位图是一个位向量。位图的大小是可变的，这个值存放在TSS段中。处理器通过TSS中的位图基址来定位这个位图。I/O位图基址是一个16位宽的字段，包含了I/O许可位图的偏移。位图的上限也是TSS段的上限。

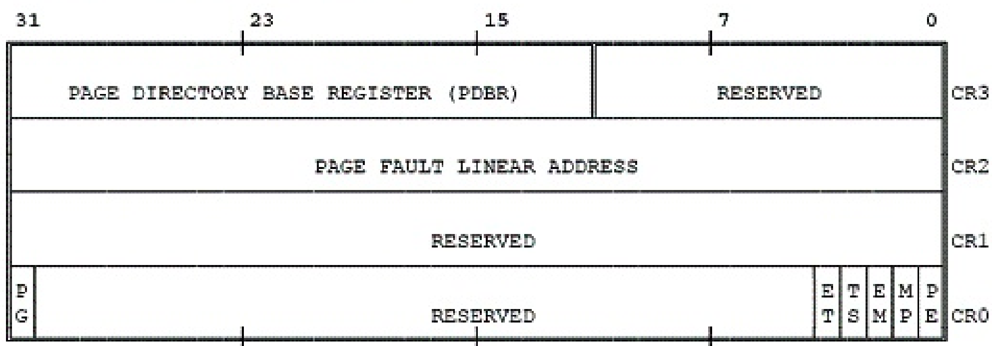
在保护模式下，当遇到一条I/O指令时（IN，INS，OUT，或OUTS），处理器首先检查是否CPL<=IOPL。如果是的话，I/O操作可以继续。如果不是的话，处理器检查I/O许可位图。（在虚拟8086模式，处理器不管IOPL而直接查看位图，参看第15章）

位图中的每一位都对应着一个I/O端口字节地址。例如，端口41的位可以在I/O位图基址+5，位偏移1，处找到。处理器会检测I/O指令访问到的每个字节，以看是否允许访问。例如，一个双字操作，将测试4位，对应着4个连续的字节地址空间。如果任一个测试是置位的，处理器引发一个通用保护异常。如果所有的测试都有为0，I/O操作被允许。

没有必要为所有的I/O地址设置I/O许可位图。没有被覆盖到的I/O地址空间，将被假设该位图对应位已设置为1。例如，如果TSS界限等于I/O位图基址+31的话，前256的端口被映射。对于更大的端口号作的I/O操作将引起异常。

如果I/O映射位图基址大于或等于TSS界限的话，TSS段则没有I/O许可位图，所以只要当CPL>IOPL时，所有的I/O操作将引起异常。

因为I/O许可位图在TSS段中，不同的任务有不同的映射位图。操作系统可以通过修改一个任务的TSS段中的I/O许可位图，来为某个任务分配端口，



第9章 异常和中断（Exceptions and Interrupts）

异常和中断是特别的控制转移方式。他们工作地象是非编程的调用一样。他们改变正常的程序流程来处理外界事件或者报告错误和异常条件。中断与异常的不同便是中断是用来处理异步的外界事件。而异常则是用来处理被处理器发现的错误。

有两种外部中断源和两种异常：

1、 中断

- 可屏蔽中断，通过INTR引脚产生。
- 不可屏蔽中断，通过NMI引脚产生。

2、 异常

- 处理器检测到的。他们被进一步分为错误（faults），陷阱(traps)，和中止(aborts)。
- 被编程的。指令INTO，INT 3，INT n, 和BOUND 能够引发异常。这些指令通常被称为“软中断”，但处理器象普通中断一样处理它们。

这一章解释了在保护模式下，处理器控制中断和对中断的反应的特性。

9.1 识别中断（Identifying Interrupts）

处理器用一个数字来标识不同类型的中断和异常。

NMI和异常的标识号已经被预先定义好了，从0~31。当前不是所有的编号都被80386使用。没有使用的标识号被INTEL用作以后扩展而保留。

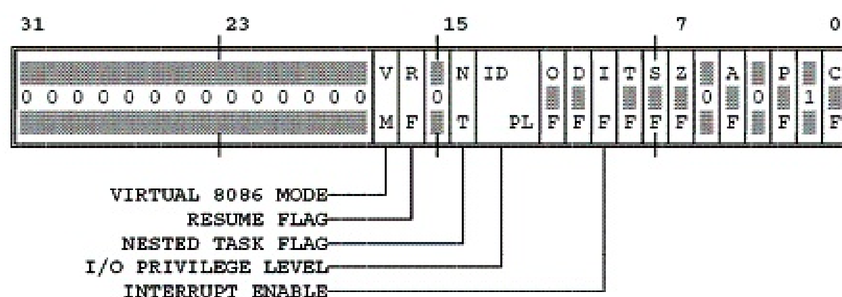
可屏蔽中断的标识号则由外部中断控制器来分配（如果8259A可编程中断控制器）当处理器的中断识别周期时和主机通信。被8259A PIC 分配的中断号可以通过软件来指定。任何一个从32到255的编号都可以使用。表9-1显示了中断和异常标识号的分配。

由它们被报告的方式和引起异常的指令是否重起（restart）,异常被分类为错误（faults），陷阱（traps），和中止（aborts）。

错误（Faults）在指令引起异常前就报告的异常是错误。错误可能在指令执行前检测到或者在指令执行期间检测到。如果是在执行期间检测到的，机器将会恢复到指令执行前的状态，以便可以重想指令。

陷阱（Traps）陷阱是在引起异常的指令边界检测到的。

中止（aborts）中止是即不能精确定位引起异常的指令也不能重起引起异常的指令的异常。中止用来报告很严重的错误，如硬件错误或系统表的不一致性和错误。



9.2 允许和禁止中断（Enabling and Disabling Interrupts）

处理器只在一条指令结束和下一条指令开始之间进行中断服务。当一个串指令有REP前缀时，中断和异常可以在每次叠代期间发生。所以长的串指令不会使处理器长时间不响应中断。

一定的条件和标志设置了以后，处理器会在指令边界禁止一些中断和异常。

9.2.1 NMI 屏蔽后来的NMIS（NMI Masks Further NMIS）

当一个NMI处理程正在执行时，处理器忽略后来的NMI引脚发送过来的中断信号，一直到下一条IRET指令被执行。

9.2.2 IF 屏蔽INTR（IF Masks INTR）

IF标志（interrupt-enable flag）控制着处理器是否接受由INTR引脚引起的外部中断。当IF=0时，INTR中断被屏蔽。当IF=1时，INTR中断被允许。和其它标志位一样，处理器在接收到一个RESET信号时，将清除IF位。CLI和STI指令用于改变IF位。

CLI（清中断允许位）和STI（设置中断允许位）显示的设置IF位（标志寄存器的位-9）。这些指令只能在CPL<=IOPL时才可以执行。如果CPL>=IOPL时，执行这些指令将引发通用保护异常。

IF被以下指令隐式的操作：

- PUSHF存储所有标志，包含IF，到堆栈上，这样他们就可以被检测了。
- 任务切换和POPF指令、IRET指令都加载标志寄存器。因此，将更改IF位。
- 通过中断门的中断将自动清除IF位，禁止中断。（这一章后面将介绍中断门）

9.2.3 RF 屏蔽调试错误（RF Masks Debug Faults）

标志寄存器中的RF位控制着调试中断的识别。这样可以在一条指令只引发一次调试中断，不管多少次的指令重起。（关于调试，参看第12章）

9.2.4 MOV 或POP 到SS将屏蔽一些中断和异常（MOV or POP to SS Masks Some Interrupts and Exceptions）

一些要改变堆栈段寄存器的软件通常会这样做：

MOV SS，AX

MOV ESP，StackTop

如果在SS加载后，而ESP加载前发生了一个中断或异常，在中断处理程序中堆栈的两部分将是不一致。

为了防止这种情况发生，80386在一条MOV 或POP向SS加载了值后，将会在指令的边界屏蔽INTR、NMI、调试异常、单步陷阱等。但一些异常还是可能发生的，例如，缺页异常和通用保护异常。如果总是使用80386的LSS指令，则不会产生这个问题。

9.3 同时发生的中断和异常的优先级（Priority Among Simultaneous Interrupts and Exceptions）

如果在一个指令边界有不止一个中断或异常挂起，处理器只能一次处理他们中的一个。中断和异常之间的优先级被表9-2显示。处理器最先处理优先级最高的中断或异常类型，把控制转移到最高优先级的中断处理程序里的第一条指令。低优先级的异常将被丢弃。低优先级的中断将被挂起。丢弃的异常将在返回到引起中断的指令处再次被发现。

9.4 中断描述符表（Interrupt Descriptor Table）

中断描述符表把每个中断或异常编号和一个指向中断处理事件服务程序的描述符联系起来。同GDT和LDT一样，IDT是一个8-字节的描述符数组。和GDT、LDT不同的是，IDT的第一项可以包含一个描述符。为了形成一个在IDT内的索引，处理器把中断、异常标识号乘以8以后来做为IDT的索引。因为只有256个编号，IDT不必包含超过256个描述符。它可以包含比256更少的项，只是那些需要使用的中断、异常的项。

IDT可以在内存的任意位置。如图9-1所示，处理器通过IDT寄存器（IDTR）来定位IDT。指令LIDT和SIDT用来操作IDTR。两条指令都有一个显示的操作数：一个6字节表示的内存地址。图9-2显示了它的格式。

LIDT（Load IDT Register）使用一个包含线性地址基址和界限的内存操作数来加载IDT。这条指令只能在特权级0执行。一般是操作系统初始化软件在创建IDT时来执行它。操作系统也可以用它来改变到另一个IDT。

SIDT（Store IDT Register）拷贝IDTR的基址和界限部分到一个内存地址。这条指令可以在任意特权级执行。

Figure 5-2. Segment Translation

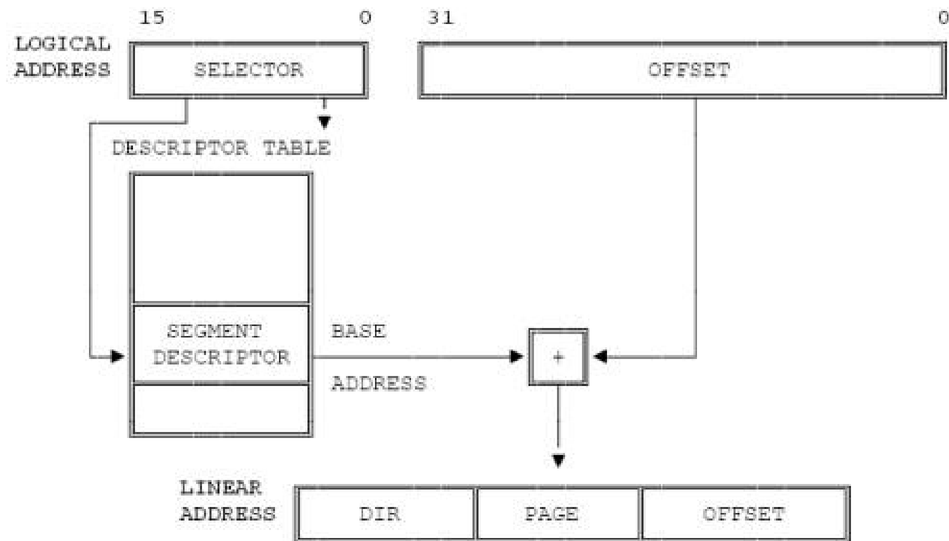
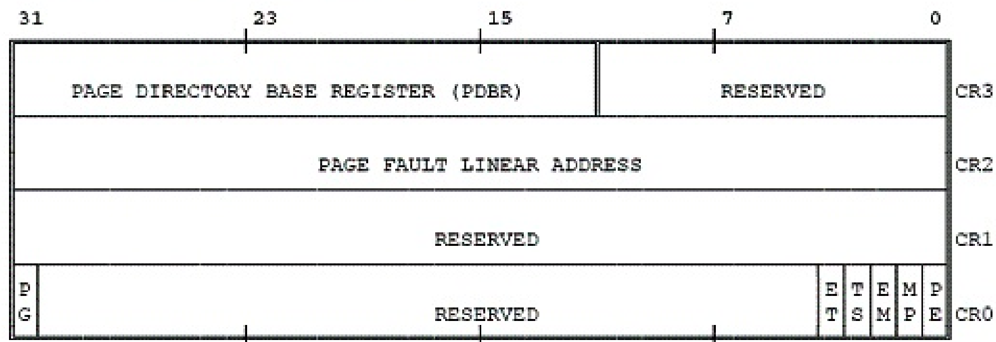


Figure 4-2. Control Registers



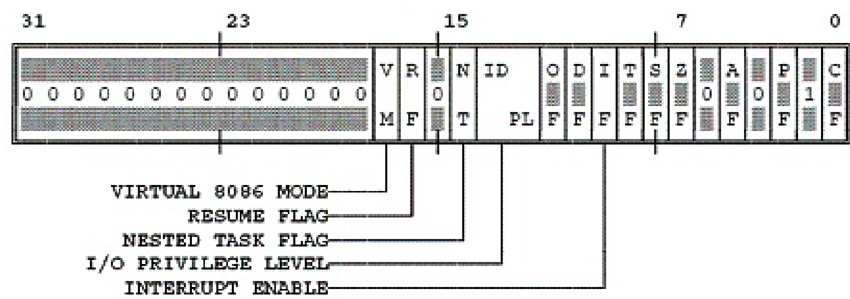
9.5 IDT 描述符（IDT Descriptors）

IDT可以包含以下三种中的任一种描述符：

- 任务门（Task gates）
- 中断门（Interrupt gates）
- 陷阱门（Trap gates）

图9-3显示了80386的任务门、中断门、陷阱门的格式。（在IDT中的任务门和在第7章讲述的任务门相同）

Figure 4-1. System Flags of EFLAGS Register



NOTE
0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE.

9.6 中断任务和中断子程序（Interrupt Tasks and Interrupt Procedures）

和CALL指令能调用一个子程序或任务一样，中断、异常也可以“调用”一个子程序或任务来做中断处理程序。当识别了一个中断、异常时，处理器使用中断号来索引IDT。如果处理器索引到的是一个中断门或陷阱门，它就象CALL指令调用一个调用门一样调用一个中断处理子程。如果处理器索引到一个任务门，它就象CALL指令调用了一个任务一样，做任务切换。

9.6.1 中断子程序（Interrupt Procedures）

如图9-4所示，中断门、陷阱门间接地指向了一个在当前任务上下文里的子程序。门里的选择子指向了一个在GDT或LDT中的可执行代码段描述符。偏移部分字段则指向了中断、异常处理子程序的入口。

80386象使用CALL指令一样唤醒一个中断、异常处理子程序。不同之处在下面介绍。

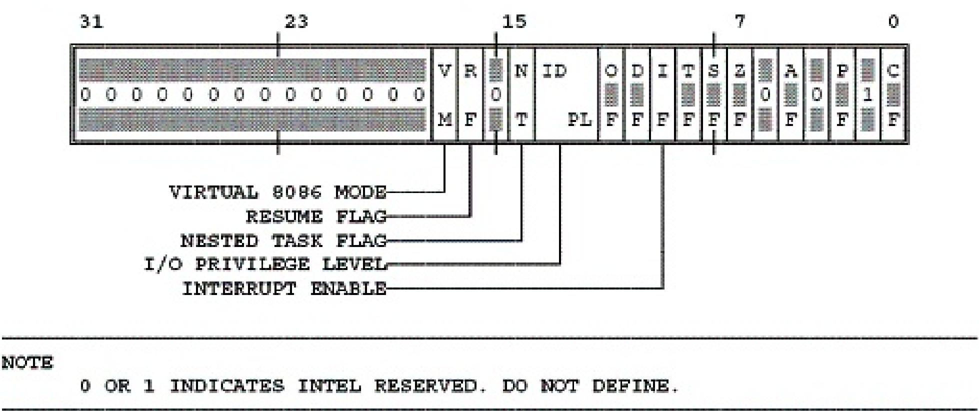


Figure 4-1. System Flags of EFLAGS Register

9.6.1.1 中断子程堆栈（Stack of Interrupt Procedure）

和CALL指令的控制转移一样，当控制转移到中断、异常处理子程序时，程序使用堆栈来存储返回被中断程序的一些信息。如图9-5所示，中断时，先把EFLAGS寄存器推入堆栈，然后再是返回地址。

某些类型的异常还可以引起一个出错码，出错码被压入堆栈。异常处理程序可以使用出错码来帮助排错。

9.6.1.2 从中断子程序返回（Returning from an Interrrupt Procedure）

中断处理程序的返回方式也不和一般的子程序相同。IRET指令用来从中断子程序中返回。IRET和RET指令相似，只是要增加EIP额外的4个字节（因为在堆栈上的标志）和把保存的标志。只有当CPL为0的时候，标志寄存器的IOPL字段才可以改变。IF标志位只有当CPL<=IOPL时，才会改变。

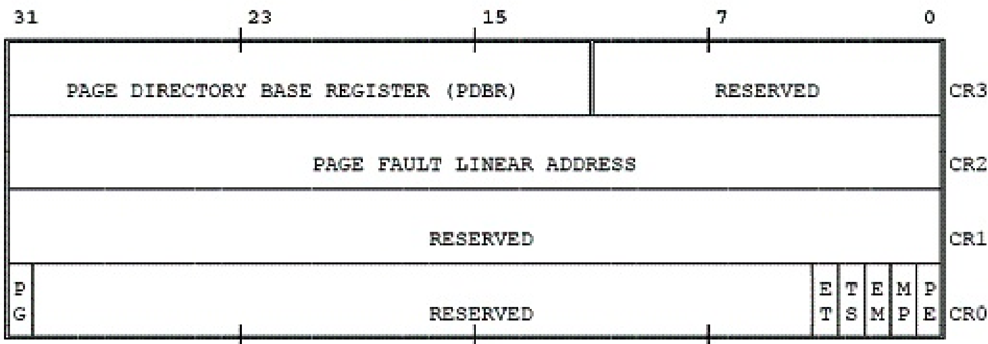


Figure 4-2. Control Registers

9.6.1.3 被中断子程使用的标志位（Flags Usage by Interrupt Procedure）

不管是中断门还是陷阱门，中断发生，且当前的TF被保存在堆栈上时都会清除TF（陷阱标志）标志位。这样，处理器就可以在中断处理程序中禁止用于调试的单步中断异常。下一个IRET指令将从堆栈上的EFLAGS寄存器映象恢复TF位。

中断门和陷阱门的主要区别是对于IF（允许中断标志）标志位的影响。通过中断门进入中断处理程序后会清除IF位，从而禁止了其它中断的发生。其后的IRET指令会恢复IF位到堆栈上的EFLAGS映象。通过陷阱门进入的中断不改变IF位。

9.6.1.4 在中断子程序内的保护（Protection in Interrupt Procedures）

对于中断子程序的特权级规则和普通的子过程调用类似：CPU不允许中断控制从当前特权级到低特权级。一个试图破坏这个规则的操作将引起通用保护异常。

因为中断的发生一般是可预测的，这种特权级的约束着中断、异常处理程序的执行。以下的方法都可以用来防止这个规则的破坏。

- 把处理程序放到一个一致性段中。这种策略可以处理一些异常（比如，除法错）。这样的处理程序只能使用堆栈上的数据。如果它需要在数据段时的数据，数据段应该为特权级3，让它不被任何保护。
- 把处理程序放在特权级0的段。

9.6.2 中断任务（Interrupt Tasks）

在IDT中的任务门间接的指向了一个任务，如图9-6所示。门里的选择子字段指向了一个GDT中的TSS描述符。

当一个IDT中的中断、异常向量指向一个任务时，任务切换发生。将中断用任务来处理有以下两个好处：

- 上下文被完整的自动保存。

- 中断处理程序可以通过一个完全隔开的地址空间，和其它任务完全隔开，通过了LDT和页目录。

处理器的任务切换操作在第7章中已讲述。中断任务通过执行一条IRET指令返回到被中断的任务。

如果任务切换是被一个带出错码的异常引起的话，处理器将自动压入出错码到中断任务的第一条指令特权级的对应的堆栈中。

当在80386中的操作系统中使用中断任务时，实现上就有两个调度器：一个软件调度器（操作系统的一部分）和一个硬件调度器（处理器中断机制的一部分）。软件调度器设计时应该考虑到，只要中断允许时，硬件调度器可能在任意时间指派中断任务。

Table 6-2. Useful Combinations of E, G, and B Bits

Case:	1	2	3	4
Expansion Direction	U	U	D	D
G-bit	0	1	0	1
B-bit	X	X	0	1
Lower bound is:				
0	X	X		
LIMIT+1			X	
shl(LIMIT,12,1)+1				X
Upper bound is:				
LIMIT	X			
shl(LIMIT,12,1)		X		
64K-1			X	
4G-1				X
Max seg size is:				
64K	X			
64K-1		X		
4G-4K			X	
4G				X
Min seg size is:				
0	X	X		
4K			X	X

shl (X, 12, 1) = shift X left by 12 bits inserting one-bits on the right

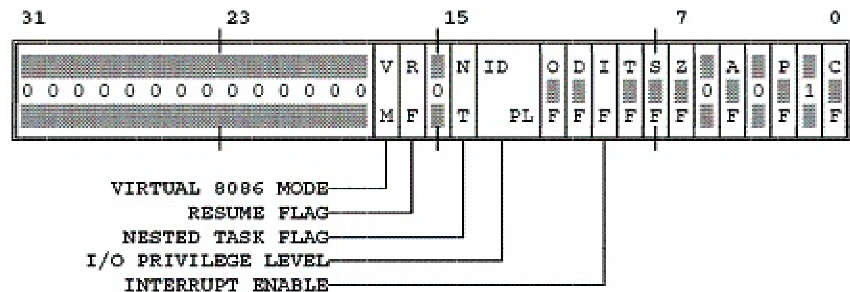
9.7 出错码（Error Code）

与某一个段相关的段的的异常的话，处理器为异常处理程序压入一个出错码到堆栈上（不管是子程序还是任务）。图9-7显示了出错码的格式。出错码的格式和选择子有点象。但是，出错码并不包含RPL字段，另外包含了2位不同的项：

1、 如果是一个程序外部的事件引起的异常，则处理器设置EXT位。

2、 如果出错码的索引部分指向了一个IDT中的门描述符，处理器设置I位。

如果I位没有设置，TI位指示了出错码是指向GDT（0值）还是指向LDT（值1）。余下的14位则是段选择子的高14位。一些情况下，在堆栈上的出错码是空的（NULL），也就是低字的所有位都是0。



9.8 异常条件（Exception Conditions）

以下几节详细的讨论可能的异常条件。每个类型都将分为错误（fault）、陷阱（trap）、或中止（abort）来讲述。这种分类有助于系统程序员重想引起异常的子程序。

错误： 当发生错误时，保存的CS和EIP将指向了引起错误异常的指令。

陷阱： 当发生陷阱时，保存的CS和EIP将指向引起陷阱的指令的下一条指令，且这下一条指令是动态的一下条。如果陷阱是在一条改变程序控制流的指令期间发生时，保存的CS和EIP则指向了控制改变后的程序的指令。例如，如果在执行一条JMP指令期间发生一个陷阱，压入堆栈的CS和EIP将是JMP的目的地指令，而不是在JMP下面的指令。

中止： 中止是不能精确定位引起异常的指令或不能重起指令的异常。中止用来报告严重的错误，如硬件出错或系统表的不一致性或错误。

9.8.1 中断0——除法错（Divide Error）

除法错误发生在当DIV或IDIV指令的除数为0时。

9.8.2 中断1—— 调试异常（Debug Exceptions）

处理器在很多情况下都会引发这个中断。这个异常是错误还是陷阱取决于以下条件：

- 指令地址断点错误（Intruction address breakpoint fault）
- 数据地址断点陷阱（Data address breakpoint trap）
- 通用错误（General detect fault）
- 单步陷阱（Single-step trap）
- 任务切换断点陷阱（Task-switch breakpoint trap）

处理器在这种情况下不会压入出错码。异常处理程序可以检察调试寄存器来决定是什么条件引起的异常。关于调试和调试寄存器，参看第12章。

9.8.3 中断3——断点（Breakpoint）

INT 3指令引发这个陷阱。INT3是一条一字节长的指令，这样就可以很容易地用断点操作码来替代代码段中操作码。操作系统或调试系统可以用一个数据段的别名来指向代码段，这样可以很方便地在任何地方放入INT3指令，来引起异常，从而进行一些进一步的操作。调试器经常在一个任务的关键地方，使用断点来显示寄存器、变量。

保存的CS：EIP值，指向了断点异常指令的下一个字节。如果一个调试器用调试操作码来替代了正常的指令，它必须要把保存的EIP减去1个字节的值。关于调试的信息，请参看第12章。

9.8.4 中断4——溢出（Overflow）

只有当处理器遇到INTO指令且OF（溢出位）标志设置时，处理器才引发这个异常。因为有符号数和无符号数使用同样的算数指令，处理器不能确定到底是哪一种操作，所以当发生溢出时，处理器并不自动引发异常。取而代之的是，如果被作为符号数处理的话，且有可能超出表示范围的话，处理器设置OF位。当对符号数操作时，仔细的程序员和编译器将自已测试OF位或使用INTO指令。

9.8.5 中断5——越界（Bounds Check）

当处理器执行BOUND指令时，且操作数超出了界限时，处理器将引发这个错误。程序可以使用BOUND指令来检察一个有符号的到指定内存区域的数组索引。

9.8.6 中断6——非法操作码（Invalid Opcode）

当执行部件遇到一条非法操作码的指令时，引发这个错误。（只有当执行到时，才会引发该异常，也就是说指令预取时取到一条非法操作数的指令时，并不会引发异常）。处理器不会压入出错码。异常可以在同一个任务中处理。

当指定类型的操作码的操作数非法时，也会引起该异常。例如，一条段间JMP去访问一个寄存器操作数、或LES指令访问一个寄存器源操作数。

9.8.7 中断7——协处理器不可用（Coprocessor Not Available）

当以下两种情况其中之一发生时，引发这个异常：

- 处理器则到一条ESC（escape）指令，且CR0（control register zero）中的EM（emulate）位设置时。
- 处理器遇到一条WAIT指令或和条ESC指令，且CR0中的MP（monitor coprocessor）位和TS（task switched）位都设置时。

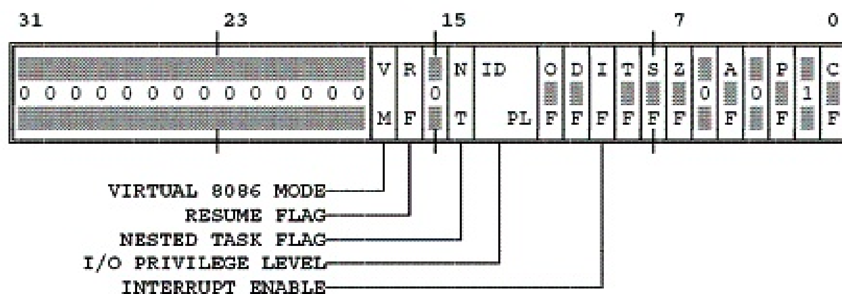
关于协处理器，请检看第11章。

9.8.8 中断8——双重错（Double Fault）

通常，当处理器刚要调用一个先前的异常的异常处理程序时又检测到一个异常，两个异常可以被连续地处理。但是，如果处理器不能串行的处理他们，处理器引发一个双重错异常。当两个错误被声明为一个双重错误时，80386把异常分为3类：良性异常（benign exceptions）、贡献异常（contributory exceptions）、和页错误（page faults），图9-3显示了这种分类。

表9-4显示了引起双重错误的异常组合。

处理器总是压入一个出错码到双重出错的异常处理程序。但是，出错码总是0。出错的指令是不可重起的。如果在将要调用双重异常处理程序时，又发生了一个异常，处理器将停机。



9.8.9 中断9——协处理器段超出（Coprocessor Segment Overrun）

处理器在保护模式执行时，当向协的中间部分传送一个协处理器的操作数到NPX时，如果检测到一个页错误或段错误时，处理器引发这个异常。

9.8.10 中断10——非法TSS（Invalid TSS）

当在任务切换时，发现新的TSS非法时，处理器发生中断10。图9-5显示了非法的TSS的情况。出错码被压入堆栈，以帮助检察错误引发的原因。EXT位指示了当前的异常是否是由程序外界引起的。也就是通过一个任务门引起的外部中断切换到这个非法的TSS。

这个错误可能发生原先的任务中或发生在新的任务的上下文中。直到处理器完全检测了新TSS的存在后，异常发生在原先任务的上下文中。当新任务的TSS存在性被检测后，任务切换就算完成了。也就是说更新了TR。如果任务切换是由于CALL指令或中断，新任务的TSS中的返回链将被设置为当前的TSS（旧的TSS）。任一个在这时之后发生的错误将在新的任务中处理。

为了能在一个任务中正确的处理这个异常，异常10应该是通过任务门在一个新的任务中处理的。

Table 9-5. Conditions That Invalidate the TSS

Error Code	Condition
TSS id + EXT	The limit in the TSS descriptor is less than 103
LTD id + EXT	Invalid LDT selector or LDT not present
SS id + EXT	Stack segment selector is outside table limit
SS id + EXT	Stack segment is not a writable segment
SS id + EXT	Stack segment DPL does not match new CPL
SS id + EXT	Stack segment selector RPL < > CPL
CS id + EXT	Code segment selector is outside table limit
CS id + EXT	Code segment selector does not refer to code segment
CS id + EXT	DPL of non-conforming code segment < > new CPL
CS id + EXT	DPL of conforming code segment > new CPL
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS segment selector is outside table limits
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS is not readable segment

9.8.11 中断11——段不存在（Segment Not Prosent）

当处理器发现一个描述符的存在位为0时，处理器引发异常11。处理器可能在以下情况下引发该错误：

- 当加载CS，DS，ES，FS，GS，寄存器时，但加载SS寄存器引发堆栈错误。
- 当用LLDT指令加载LDT寄存器时。在任务切换时加载LDT寄存器，则引发非法

TSS异常。

- 当使用一个不存在的门描述符时。

这样的错误是可以重起的。如果异常处理程序把段的存在位设置且返回后，被中断的程序将继续执行。

如果段不存在异常发生在任务切换过程中，任务切换的步骤没有完全完成。在任务切换的过程中，处理器首先加载所有的段寄存器，然后检察它们内容的有效性。如果一个段不存在异常被检测到，那么剩下的段寄存器的值将是未经过检测的，所以可能是不可用来访问内存的。异常处理程序不应该在没有引起另一个异常前依赖于此时的CS，SS，DS，ES，FS和GS。异常处理程序应该在恢复新任务之前首先检测所在段寄存器。否则，通用保护异常可能会随后发生，以致错误检测将更加困难。有3种方法来处理这种情况：

1、 在一个任务中处理段不存在异常。当切换回被中断的任务时，处理器将从TSS加载时检测寄存器的有效性。

2、 压入和弹出所有段寄存器。每一条POP指令将引起处理器检测段寄存器的新内容。

3、 细查TSS段中存储的每一个段寄存器映象，模拟处理器在加载段寄存器时的检测。

这个异常压入一个出错码到堆栈上。EXT位指出是否是外部事件引起的段不存在的异常。如果出错码访问的是IDT的项，I位将被设置，也就是说一条INT指令访问了一个不存在的门。

操作系统通常使用“段不存在”异常来实现基于段的虚拟内存。但是，一个门描述中的不存位，通常不是指段的不存在（因为门不一定要对应着一个段）。门描述符的不存在可以被操作系用来引发一个特别重要的异常。

9.8.12 中断12——堆栈异常（Stack Exception）

堆栈异常通常发生在以下两种情况下：

- 在使用SS寄存器来访问内存时，如果发生了任何的界限违例。这包括了基于堆栈的指令，如POP，PUSH，ENTER，还有LEAVE，当然还有其它的一些隐式使用SS的内存访问（例如，MOV AX， [BP+6]）。当堆栈太小而不能容纳指定的局部变量时，ENTER指令将引起这个异常。
- 当加载一个选择子到SS寄存器时，且该选择子指向一个标识为不存在但有效的描述符。这种情况可能发生在任务切换中、段间CALL指令、段间返回、LSS指令、或者一条向SS加载的MOV或POP指令。

当处理器发现堆栈异常时，它会压入一个出错码到异常处理程序的堆栈上。如果异常是由堆栈段不存在或在段间CALL指令间时的新堆栈溢出的话，出错码包含了出问题的段的选择子（异常处理程序可以测试描述符的存在位来确定是哪个异常发生的）。否则出错码为0。

造成这种异常的指令在所有情况下都是可重起的。被压入入异常处理程序堆栈的返回地址指向了一条需要重起的指令处，一般来说就是引起异常的指令。但是，在一个任务切换过程中，加载不存在的堆栈段寄存器时，它指向了新任务的第一条指令。

当堆栈错误在任务切换时发生的话，段寄存器不能再用来内存访问了。在任务切换中，选择子是在描述符被检察之前加载到寄存器里的，所以可能并不能用来作内存访问。堆栈异常处理程序不应在未引起另一个异常之前依赖于CS，SS，DS，ES，FS和GS中的值。异常处理程序应该要在重起任务前检测所有的段寄存器的值。否则，通用保护异常错误将会使以后的错误调试更加困难。

9.8.13 中断13——通用保护异常（General Protection Exception）

所有保护模范规则的违例，如果没有引起另一个异常，将引起一个通用保护异常。这些包括（但不局限于）：

1、 当使用CS，DS，ES，FS，或GS，做内存访问时的段界限超出。

2、 访问描述符表时的界限超出。

3、 向一个不可执行的段作控制转移。

4、 向一个只读段或一个代码段写入数据。

5、 从只执行的代码段内读取数据。

6、 用一个指向只读描述符的选择子加载SS寄存器（除非选择子来自于任务切换过程时的TSS段中，这种情况将引发一个TSS异常）

7、 把系统段描述符加载到SS，DS，ES，FS，或GS。

8、 把一个不可读的执行代码段描述符加载到DS，ES，FS或GS中。

9、 将代码段描述符加载到SS寄存器。

10、 DS，ES，FS，和GS中包含一个空选择子（null selector）来访问内存时。

11、 向一个正忙的任务切换。

12、 违反特权级规则。

13、 把一个PG=1而PE=0的值加载到CR0内。

14、 通过中断门或陷阱门从V86模式转移到不是特权级0时。

15、 执行一个长度大于15个字节的指令（只可能发生在达多的前缀用于一条指令前时）。

通用保护异常是一个错误。在响应这个异常时，处理器压入一个出错码到异常处理程序的堆栈上。如果在加载一个描述符时，发生异常，出错码包含了指向此描述符的选择子。否则，出错码为空。出错码中的选择子可能来自以下：

1、 指令操作数。

2、 一个指令操作数中的门，选择子在门中。

3、 在任务切换过程中，在TSS中的选择子。

9.8.14 中断14——缺页异常（Page Fault）

当启用了分页后（PG=1），当处理器将线性地址转换到物理地址时，检测到以下情况中的一个将引发一个异常：

- 所需要的页目录或页表项的存在位为0时。
- 当前子程序没有足够的权限来访问指定的页面。

处理器为缺页异常处理程序提供以下两种信息以便异常的诊断和从异常中恢复：

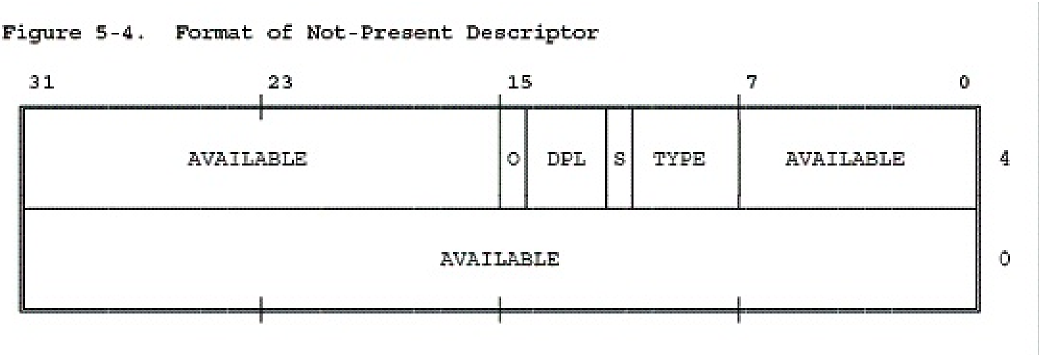
- 一个在堆栈上的出错码。为缺页异常提供的出错码和一般的异常的出错码格式有所不同（见图9-8）。出错码告诉异常处理程序以下三件事：

1、引起异常的原因是由于页不存在还是没有足够的权限来访问指定的页。

2、异常发生时，处理器是处于用户模式还是处于超级用户模式。

3、在访问内存时，是读操作还是写操作。

- CR2 （控制寄存器2）。处理器把引起异常的线性地址放在CR2中（如图9-9）。异常处理程序可以使用这个线性地址来定位页目录项和页表项。如果在这个异常处理过程中，允许另一个缺页异常产生的话，异常处理程序应该负责把CR2压入到堆栈中。



9.8.14.1 在任务切换中的缺页异常（Page Fault During Task Switch）

在任务切换时，处理器可能访问以下4个段：

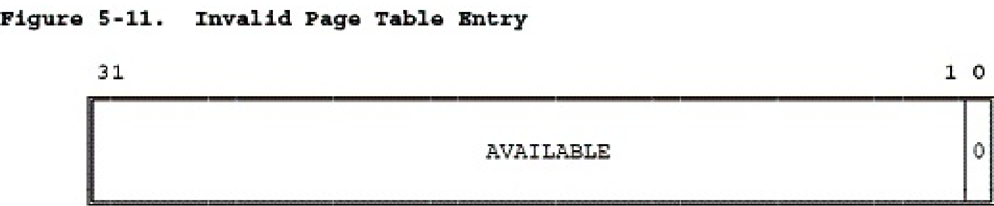
1、 把当前的任务状态写入到它的任务状态段中。

2、 读取GDT来定位新任务的TSS描述符。

3、 读取新任务的TSS，以便检测段描述符的类型。

4、 可能会读取新任务的LDT，以便来检测存储在新任务TSS中段寄存器。

当访问他们中的任意一个段时，都可能出现缺页异常。在后两种情况下，异常算发生在新任务的上下文里。保存的指针指向新任务的下一条指令，而不是引起任务切换的指令。如果操作系统的设计允许在任务切换时发生缺页异常，缺页异常错误应该通过一个任务门来处理。



9.8.14.2 缺页错误内的不一致堆栈指针（Page Fault with Inconsistent Stack）

为了保证在缺页异常中不会让处理器使用非法的堆栈指针（SS：ESP），应该特别注意。在80386早期写的软件常常使用一对指令还改变堆栈，如：

MOV SS，AX

MOV SP，StackTop

在80386下，第二条指令要访问内存，可能会在SS改变后而在SP改变前发生缺页异常。这时，堆栈的两部分SS：SP（或，对于32位程序来说，SS：ESP）将不一致。

如果在处理缺页异常时，发生了堆栈切换到一个定义好的堆栈（也就是说处理程序是一个任务或是一个特权级更高的子程序）的话，处理器就不会使用不一致的堆栈指针。即使这样，如果缺页异常是在一个陷阱门或中断门时处理的，且缺页异常处理程序和发生缺页异常的程序是在同一特权级的话，处理器会使用当前的（非法的）堆栈指针。

在实现分页和缺页异常处理程序在同一任务内（用陷阱门或中断门）的系统里，同特权级的软件应该用新的LSS指令，而不要使用一对上面那样的指令，来初始化堆栈。当缺页异常处理程序在特权级0（正常情况下应该是）执行时，问题则只局限在特权级0的代码，一般说来是操作系统内核。

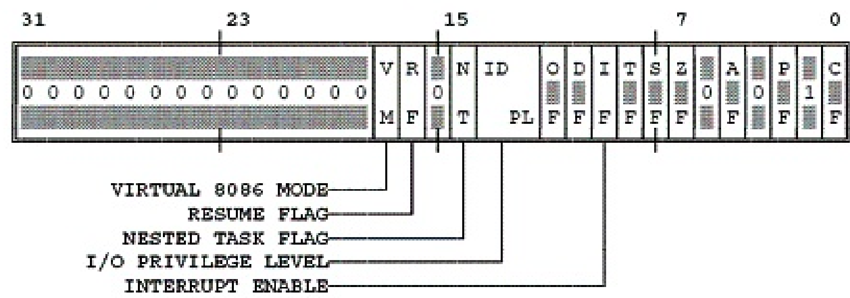
9.8.15 中断16——协处理器错（Coprocessor Error）

当处理器从ERROR#引脚发现一个80287或80387发送的一个报告时，处理器引发这个异常。80386只在一定情况下的ESC指令或者当遇到WAIT指令且MSW中的EM位为0时（没有摸拟）执行前才检测这个引脚。关于协处理器的信息，请参看第11章。

9.9 异常总结（Exception Summary）

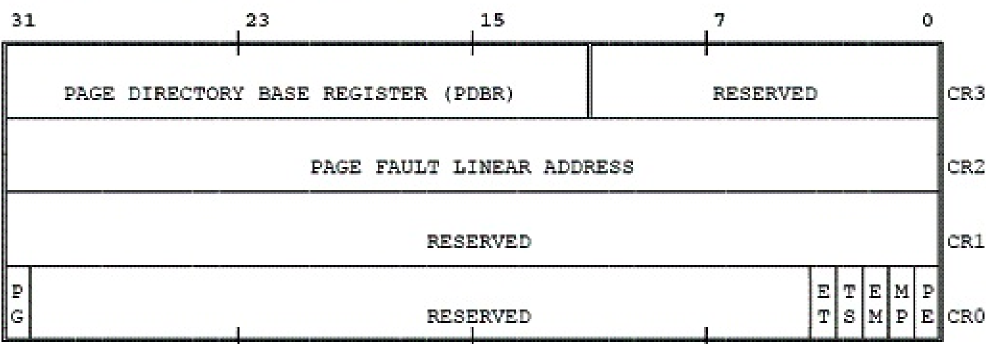
表9-6总结了被386识别的异常。

Figure 4-1. System Flags of EFLAGS Register



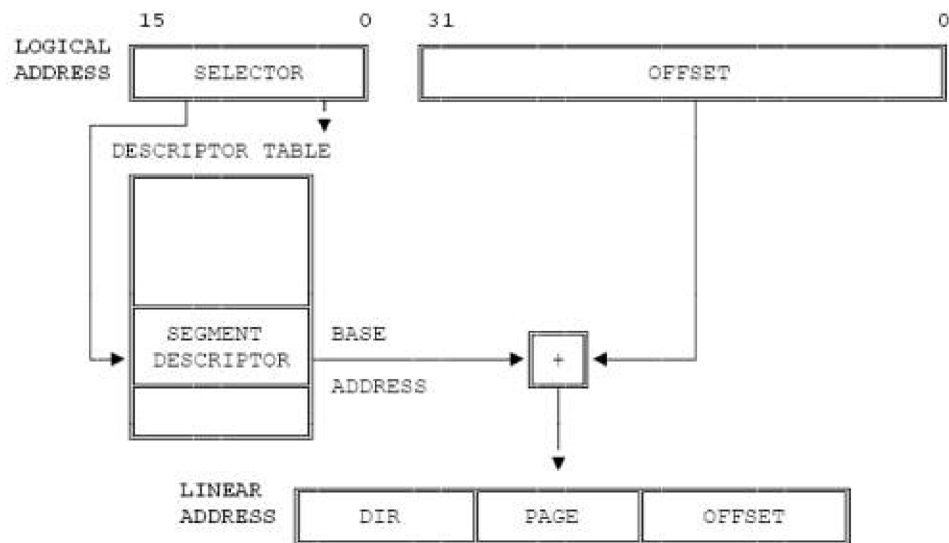
NOTE
0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE.

Figure 4-2. Control Registers



9.10 出错码总结（Error Code Summary）

表9-7总结了对于每个异常可能的出错码信息。



第10章 初始化（Initialization）

当RESET引脚接收到信号后，80386处理机的某些寄存器将被设置成预先定义好的值。这些值可以完成一个自举程序（bootstrap program）的执行，但在能完全使用处理器的特性前，还需要软件的进一步初始化。

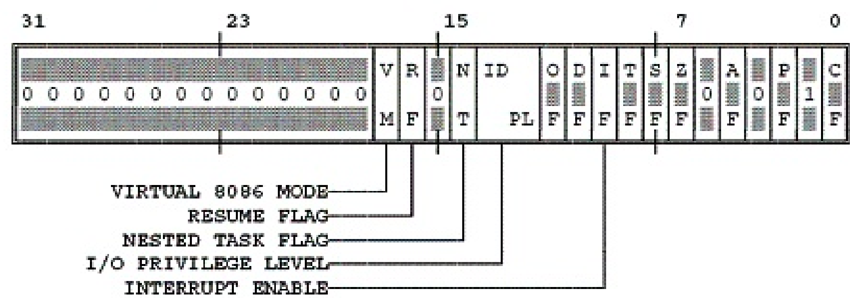
10.1 复位后处理器状态（Processor State After Reset）

EAX寄存器的值将取决于加电自检的结果。自检程序可能还要在RESET后有一个BUSY信号的高电位。如果通过自检，EAX值为0。非0值意为着处理器某些部件出了故障。如果没作自检，EAX里的值没有定义。

如图10-1所示，在复位后，DX寄存器包含了组件的标识号和修定编号。DH包含3，则指定了80386组件。DL包含一个唯一的修定号。

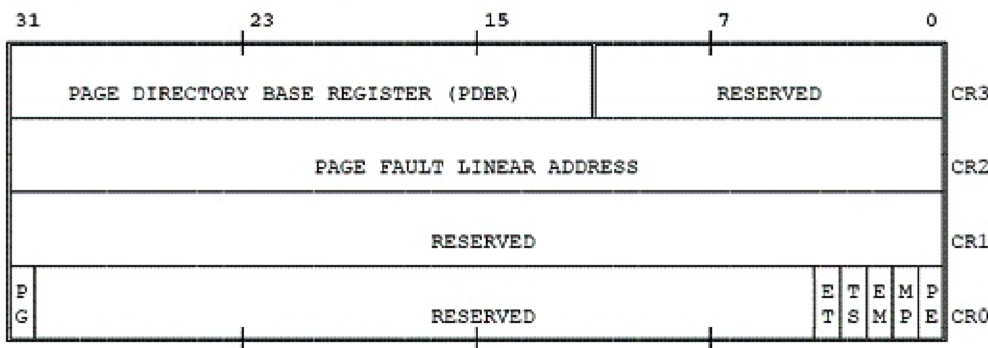
图10-2显示了，控制寄存器0（CR0）包含的值。如果当前配置下（复位后ERROR引脚的状态），80387存在的话，CR0的ET位被设置。如果ET清除，则可能没有协处理器或只包含一个80287协处理器。软件应该把后来这两种可能性区别开来。

余下的寄存器和标志位如下所示：



以上未提到的寄存器的值，都是未定义的。

这些设置说明了，处理器开始时工作在实模式，且禁止中断。



10.2 实模式初始化（Software Initialization for Real-Address Mode）

在实模式下，当程序可以完全使用处理器的特性前，一些数据结构必须先初始化。

10.2.1 堆栈（Stack）

当堆栈段寄存器（SS寄存器）没有被加载前，所有使用堆栈的指令都是不可以使用的。SS必须指向RAM内的一片区域。

10.2.2 中断表（Interrupt Table）

80386开始时是禁止中断的。但是，当一个异常或是不可屏蔽中断（NMI）发生时，处理器还是可能访问中断表。初始化软件应做一个以下的操作：

- 把IDTR中的界限值改为0。这可以使，当发生异常或不可屏蔽中断时，处理器停机。（关于如何在外部引起处理器的停机，请参看80386硬件参考手册）
- 将可能使用到的异常或中断的有效的中断处理程序的指针放入中断表中。
- 改变IDRT，使之指向一个合法的中断表。

10.2.3 第一条指令（First Instructions）

复位以后，地址线{31-20}将自动置成高电平，以便取指令。这样，加之CS：IP的初值，导致指令从物理地址 FFFFFFF0H处开始执行。近（段内的）的控制转移可以用来将控制转移到高于这个地址空间的64K字节的某处。第一条远的（段间的）ＪＭＰ或ＣＡＬＬ指令将让Ａ｛３１－２０｝置成低电位，所以８０３８６处理器就又执行在物理内存的低１Ｍ字节内了。这样的自动设置地址线Ａ｛３１－２０｝允许系统设计者把高端地址空间设计成ＲＯＭ，从而来初始化系统。

10.3 切换到保护模式（Switching to Protected Mode）

把MSW的PE位（CR0内），将使80386工作在保护模式中。起始的当前特权级为0。段寄存器和在实模式下指向了相同的线性地址（在实模式中，线性地址和物理地址相同）。

在设置了PE位以后，初始代码要立即执行一条JMP指令，以刷新处理器预取指令队列。80386会在使用前预取、解码指令和地址。但是，当切换到保护模式时，预取的指令将不再有效（属于实模式的）。JMP指令将会使处理器罢弃无效的信息。

10.4 保护模式初始化（Software Initialization for Protected Mode）

许多保护模式所需要的初始化即可以在进入保护模式之前做，也可以在进入保护模式以后做。如果在进入保护模式以后做，那么软件就不应该使用还没有初始化的保护模式特性。

10.4.1 中断描述符表（Interrupt Descriptor Table）

IDTR即可以在实模式也可以在保护模式加载。保护模式的中断表和实模式的中断表的格式是不相同的。改变到保护模式的同时也改变中断表的格式是不可能的。所以，如果IDTR指向了一个中断表，在某些时候表的格式是错误的，这种情况不可避免。在这个时候发生的中断或异常将会导致不可预测的结果。为了防止这种不可预测性，应该把保护模式的中断处理程放到IDT中后，才开启中断。

10.4.2 堆栈（Stack）

SS寄存器可以在保护模式加载，也可以在实模式加载。如果在实模式加载，当切换到了保护模式以后SS将继续指向相同的线性地址基址处。

10.4.3 全局描述符表（Global Descriptor Table）

在保护模式下，在任何段寄存器改变前，GDT寄存器必须指向一个有效的GDT。GDT和GDTR的初始化可以在实模式下完成。GDT（还有所有的LDT）应该要在RAM内，因为处理器将修改描述符表的已访问位（accessed bit）。

10.4.4 页表（Page Tables）

页表和页目录基址寄存器（CR3中）可以在实模式初始化，也可以在保护模式中。但是，CR0中的允许分页位（PG位）不能在处理器未处于保护模式时设置。PG可以和PE同时设置，或者更后。当PG设置时，CR3中的PDBR应该已经过有效的初始化，指向物理内存中的有效的页目录。初始化子程可以使用以下的方针来何证分页前后的地址的一致性：

- 在分页前后，当前被执行的代码页应该被映射到相同的物理地址。
- 当设置了PG位后，立即执行一条JMP指令。

10.4.5 第一个任务（First Task）

初始化子程可以在没有初始化任务寄存器前，在保护模式下运行一段时间。但是，当第一个任务切换发生时，以下条件必须要满足：

- 新任务必须要有一个有效的任务状态段（TSS）。在这个TSS中的比当前任务特权级高或相等的特权级的堆栈指针必须要指向有效的堆栈区。
- 任务寄存器必须指向一个区域以保存当前任务的状态。第一个任务切换以后，这些转存的信息可以不需要，这片区域也可以用作别的目的。

10.5 初始化示例

```
$TITLE ('Initial Task')
NAME INIT
init_stack SEGMENT RW
DW 20 DUP(?)
tos LABEL WORD
init_stack ENDS
init_data SEGMENT RW PUBLIC
DW 20 DUP(?)
init_data ENDS
init_code SEGMENT ER PUBLIC
ASSUME DS:init_data
nop
nop
nop
init_start:
; set up stack
mov ax, init_stack
mov ss, ax
mov esp, offset tos
mov a1,1
blink:
xor a1,1
out 0e4h,a1
mov cx,3FFFh
here:
dec cx
jnz here
jmp SHORT blink
hlt
init_code ends
END init_start, SS:init_stack, DS:init_data
$TITLE('Protected Mode Transition -- 386 initialization')
NAME RESET
;*****************************************************************
; Upon reset the 386 starts executing at address 0FFFFFFF0H. The
; upper 12 address bits remain high until a FAR call or jump is
; executed.
;
; Assume the following:
;
;
; - a short jump at address 0FFFFFFF0H (placed there by the
;     system builder) causes execution to begin at START in segment
;     RESET_CODE.
;
;
; - segment RESET_CODE is based at physical address 0FFFF0000H,
;     i.e. at the start of the last 64K in the 4G address space.
;     Note that this is the base of the CS register at reset. If
;     you locate ROMcode above this address, you will need to
;     figure out an adjustment factor to address things within this
;     segment.
;
;*****************************************************************
$EJECT ;
; Define addresses to locate GDT and IDT in RAM.
; These addresses are also used in the BLD386 file that defines
; the GDT and IDT. If you change these addresses, make sure you
; change the base addresses specified in the build file.
GDTbase EQU 00001000H ; physical address for GDT base
IDTbase EQU 00000400H ; physical address for IDT base
PUBLIC GDT_EPROM
PUBLIC IDT_EPROM
PUBLIC START
DUMMY segment rw ; ONLY for ASM386 main module stack init
```

```
DW 0
DUMMY ends
;*****************************************************************
;
; Note: RESET CODE must be USEl6 because the 386 initally executes
; in real mode.
;
RESET_CODE segment er PUBLIC USE16
ASSUME DS:nothing, ES:nothing
;
; 386 Descriptor template
DESC STRUC
lim_0_15 DW 0 ; limit bits (0..15)
bas_0_15 DW 0 ; base bits (0..15)
bas_16_23 DB 0 ; base bits (16..23)
access DB 0 ; access byte
gran DB 0 ; granularity byte
bas_24_31 DB 0 ; base bits (24..31)
DESC ENDS
; The following is the layout of the real GDT created by BLD386.
; It is located in EPROM and will be copied to RAM.
;
; GDT[O] ... NULL
; GDT[1] ... Alias for RAM GDT
; GDT[2] ... Alias for RAM IDT
; GDT[2] ... initial task TSS
; GDT[3] ... initial task TSS alias
; GDT[4] ... initial task LDT
; GDT[5] ... initial task LDT alias
;
; define entries in GDT and IDT.
GDT_ENTRIES EQU 8
IDT_ENTRIES EQU 32
; define some constants to index into the real GDT
GDT_ALIAS EQU 1*SIZE DESC
IDT_ALIAS EQU 2*SIZE DESC
INIT_TSS EQU 3*SIZE DESC
INIT_TSS_A EQU 4*SIZE DESC
INIT_LDT EQU 5*SIZE DESC
INIT_LDT_A EQU 6*SIZE DESC
;
; location of alias in INIT_LDT
INIT_LDT_ALIAS EQU 1*SIZE DESC
;
; access rights byte for DATA and TSS descriptors
DS_ACCESS EQU 010010010B
TSS_ACCESS EQU 010001001B
;
; This temporary GDT will be used to set up the real GDT in RAM.
Temp_GDT LABEL BYTE ; tag for begin of scratch GDT
NULL_DES DESC <> ; NULL descriptor
FLAT_DES DESC <0FFFFH,0,0,92h,0CFh,0>
GDT_eprom DP ? ; Builder places GDT address and limit
; in this 6 byte area.
IDT_eprom DP ? ; Builder places IDT address and limit
; in this 6 byte area.
;
; Prepare operand for loadings GDTR and LDTR.
TGDT_pword LABEL PWORD ; for temp GDT
DW end_Temp_GDT_Temp_GDT -1
DD 0
GDT_pword LABEL PWORD ; for GDT in RAM
DW GDT_ENTRIES * SIZE DESC -1
DD GDTbase
IDT_pword LABEL PWORD ; for IDT in RAM
DW IDT_ENTRIES * SIZE DESC -1
DD IDTbase
end_Temp_GDT LABEL BYTE
;
; Define equates for addressing convenience.
GDT_DES_FLAT EQU DS:GDT_ALIAS +GDTbase
IDT_DES_FLAT EQU DS:IDT_ALIAS +GDTbase
```

```
INIT_TSS_A_OFFSET EQU DS:INIT_TSS_A
INIT_TSS_OFFSET EQU DS:INIT_TSS
INIT_LDT_A_OFFSET EQU DS:INIT_LDT_A
INIT_LDT_OFFSET EQU DS:INIT_LDT
; define pointer for first task switch
ENTRY POINTER LABEL DWORD
DW 0, INIT_TSS
;*******************************************************************
;
; Jump from reset vector to here.
START:
CLI ;disable interrupts
CLD ;clear direction flag
LIDT NULL_des ;force shutdown on errors
;
; move scratch GDT to RAM at physical 0
XOR DI,DI
MOV ES,DI ;point ES:DI to physical location 0

MOV SI,OFFSET Temp_GDT
MOV CX,end_Temp_GDT-Temp_GDT ;set byte count
INC CX
;
; move table
REP MOVS BYTE PTR ES:[DI],BYTE PTR CS:[SI]

LGDT tGDT_pword ;load GDTR for Temp. GDT
;(located at 0)

; switch to protected mode

MOV EAX,CR0 ;get current CRO
MOV EAX,1 ;set PE bit
MOV CRO,EAX ;begin protected mode
;
; clear prefetch queue
JMP SHORT flush
flush:
; set DS,ES,SS to address flat linear space (0 ... 4GB)

MOV BX,FLAT_DES-Temp_GDT
MOV US,BX
MOV ES,BX
MOV SS,BX
;
; initialize stack pointer to some (arbitrary) RAM location
MOV ESP, OFFSET end_Temp_GDT
;
; copy eprom GDT to RAM
MOV ESI,DWORD PTR GDT_eprom +2 ; get base of eprom GDT
; (put here by builder).
MOV EDI,GDTbase ; point ES:EDI to GDT base in RAM.
MOV CX,WORD PTR gdt_eprom +0 ; limit of eprom GDT
INC CX
SHR CX,1 ; easier to move words
CLD
REP MOVS WORD PTR ES:[EDI],WORD PTR DS:[ESI]
;
; copy eprom IDT to RAM
;
MOV ESI,DWORD PTR IDT_eprom +2 ; get base of eprom IDT
; (put here by builder)
MOV EDI,IDTbase ; point ES:EDI to IDT base in RAM.
MOV CX,WORD PTR idt_eprom +0 ; limit of eprom IDT
INC CX
SHR CX,1
CLD
REP MOVS WORD PTR ES:[EDI],WORD PTR DS:[ESI]
; switch to RAM GDT and IDT
;
LIDT IDT_pword
LGDT GDT_pword
```

```
;
MOV BX,GDT_ALIAS ; point DS to GDT alias
MOV DS,BX
;
; copy eprom TSS to RAM
;
MOV BX,INIT_TSS_A ; INIT TSS A descriptor base
; has RAM location of INIT TSS.
MOV ES,BX ; ES points to TSS in RAM
MOV BX,INIT_TSS ; get inital task selector
LAR DX,BX ; save access byte
MOV [BX].access,DS_ACCESS ; set access as data segment
MOV FS,BX ; FS points to eprom TSS
XOR si,si ; FS:si points to eprom TSS
XOR di,di ; ES:di points to RAM TSS
MOV CX,[BX].lim_0_15 ; get count to move
INC CX
;
; move INIT_TSS to RAM.
REP MOVS BYTE PTR ES:[di],BYTE PTR FS:[si]
MOV [BX].access,DH ; restore access byte
;
; change base of INIT TSS descriptor to point to RAM.
MOV AX,INIT_TSS_A_OFFSET.bas_0_15
MOV INIT_TSS_OFFSET.bas_0_15,AX
MOV AL,INIT_TSS_A_OFFSET.bas_16_23
MOV INIT_TSS_OFFSET.bas_16_23,AL
MOV AL,INIT_TSS_A_OFFSET.bas_24_31
MOV INIT_TSS_OFFSET.bas_24_31,AL
;
; change INIT TSS A to form a save area for TSS on first task
; switch. Use RAM at location 0.
MOV BX,INIT_TSS_A
MOV WORD PTR [BX].bas_0_15,0
MOV [BX].bas_16_23,0
MOV [BX].bas_24_31,0
MOV [BX].access,TSS_ACCESS
MOV [BX].gran,O
LTR BX ; defines save area for TSS
;
; copy eprom LDT to RAM
MOV BX,INIT_LDT_A ; INIT_LDT_A descriptor has
; base address in RAM for INIT_LDT.
MOV ES,BX ; ES points LDT location in RAM.
MOV AH,[BX].bas_24_31
MOV AL,[BX].bas_16_23
SHL EAX,16
MOV AX,[BX].bas_0_15 ; save INIT_LDT base (ram) in EAX
MOV BX,INIT_LDT ; get inital LDT selector
LAR DX,BX ; save access rights
MOV [BX].access,DS_ACCESS ; set access as data segment
MOV FS,BX ; FS points to eprom LDT
XOR si,si ; FS:SI points to eprom LDT
XOR di,di ; ES:DI points to RAM LDT
MOV CX,[BX].lim_0_15 ; get count to move
INC CX
;
; move initial LDT to RAM
REP MOVS BYTE PTR ES:[di],BYTE PTR FS:[si]
MOV [BX].access,DH ; restore access rights in
; INIT_LDT descriptor
;
; change base of alias (of INIT_LDT) to point to location in RAM.
MOV ES:[INIT_LDT_ALIAS].bas_0_15,AX
SHR EAX,16
MOV ES:[INIT_LDT_ALIAS].bas_16_23,AL
MOV ES:[INIT_LDT_ALIAS].bas_24_31,AH
;
; now set the base value in INIT_LDT descriptor
MOV AX,INIT_LDT_A_OFFSET.bas_0_15
MOV INIT_LDT_OFFSET.bas_0_15,AX
MOV AL,INIT_LDT_A_OFFSET.bas_16_23
```

```
MOV INIT_LDT_OFFSET.bas_16_23,AL
MOV AL,INIT_LDT_A_OFFSET.bas_24_31
MOV INIT_LDT_OFFSET.bas_24_31,AL
;
; Now GDT, IDT, initial TSS and initial LDT are all set up.
;
; Start the first task!
'
JMP ENTRY_POINTER
RESET_CODE ends
END START, SS:DUMMY,DS:DUMMY
```

10.6 TLB测试

80386提供了一种机制来测试转换后备缓冲区（TLB）,该缓冲区用来把线性地址转换成物理地址。尽管TLB硬件错误的机会非常小，但用户可能在上电信心测试的时候把TLB信心测试包含进来。

注意：

TLB测试机制是80386独有的，可能不会在将来的处理器中包含它。使用这种机制的软件可能会与将来的处理器不兼容。

当测试TLB的时候，建议您关闭页（CR0中的PG=0）以避免在正在写入TLB的数据引起冲突。

10.6.1 TLB结构

TLB是一个4路关联接收机。图10－3说明了TLB的结构。它有4个装置，每个包含8项。每项包含标签和数据。标签24位宽，它们包含高阶20位线性地址，检查位，以及3个属性位。数据域包含高阶20位物理地址。

10.6.2 测试寄存器

两个寄存器，如图10－4，用来测试。TR6是测试命令寄存器，TR7是测试数据寄存器。这些寄存器可以用MOV命令来访问。测试命令寄存器可以是源操作数，也可以是目的操作数。MOV在实地址模式和保护模式下都有定义。测试寄存器是特权资源；在保护模式下，只能在特权级0用MOV访问它们。在其他特权级的访问将导致通用保护异常。

测试命令寄存器（TR6）包含一个命令位和地址标签：

C

命令位。有两个TLB测试命令：向TLB写，和TLB检查。为了向TLB写数据，可以将双字用MOV写入TR6，并将该标志清零。为了TLB检查，可以将双字用MOV写入TR6，并且置位该标志位。

Linear Address

当执行TLB写后，一个TLB项被定位到这个线性地址；这个TLB项的其他部分由TR7和刚刚被写入的TR6来决定。当执行TLB检查时，通过这个值来审查TLB；如果有且仅有一项匹配，TR6和TR7的其他部分由该匹配项来设置。

V

TLB项的检查位。TLB用这项来检查TLB项是否含有合法数据。没有被分配数据的TLB项该标志位为零。所有的检查位可以通过写CR3来清除。

D, D#

脏标志（和它的补）来设置/读取TLB项。

U, U#

U/S标志（和它的补）来设置/读取TLB项。

W, W#

R/W标志（和它的补）来设置/读取TLB项。

这些标志对的含义见表10－1，X代表D, U, 或者W。

测试寄存器（TR7）用保存从TLB读取的数据，或要写入TLB的数据。

physical Address

TLB的数据域。当TLB写后，被分配给TR6中线性地址的TLB项被设置成这个值。当执行TLB检查时，如果设置了HT，TLB的数据域（物理地址）被读到这里。如果没有设置HT，该域未定义。

HT

对于TLB检查，HT决定检查被触发（HT<-1）或忽略（HT<-0）。对于TLB写，HT必须为1。

REP

对于TLB写，选择4个通道中的一个来写入。对于TLB读，如果设置了HT，REP报告出哪个通道找到了标签；如果没有设置HT，REP未定义。

Table 10-1. Meaning of D, U, and W Bit Pairs

```
X X# Effect during Value of bit X
TLB Lookup     after TLB Write
0 0 (undefined) (undefined)
0 1 Match if X=0 Bit X becomes 0
1 0 Match if X=1 Bit X becomes 1
1 1 (undefined) (undefined)
```

Figure 10-3. TLB Structure

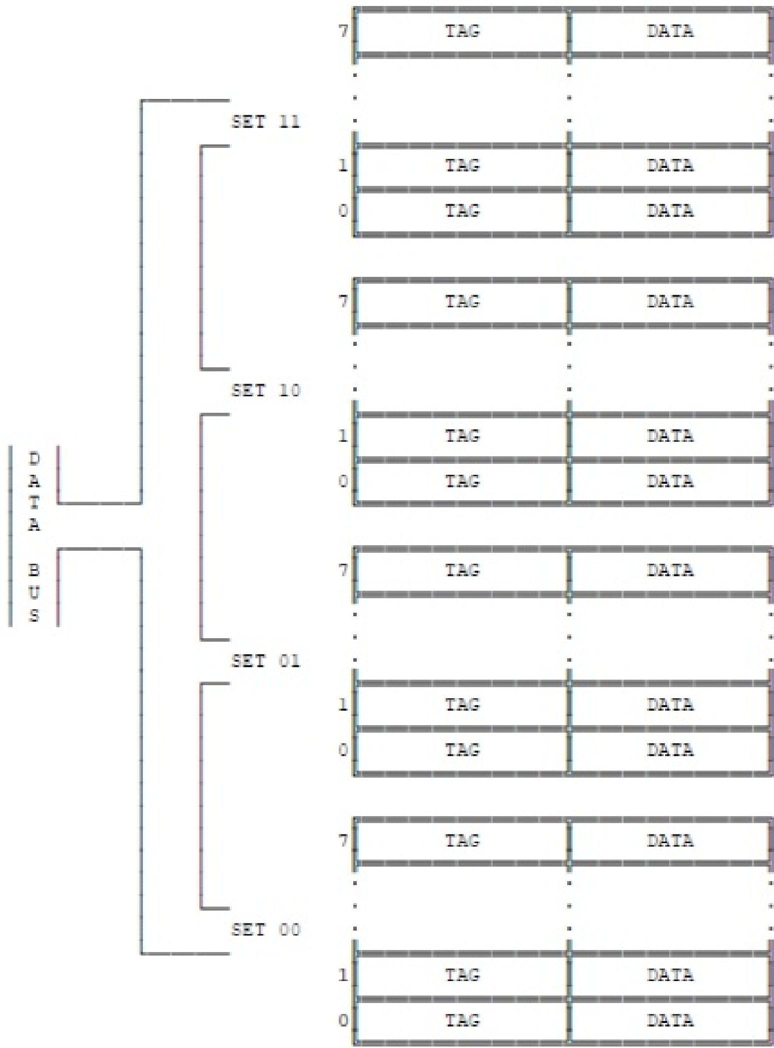
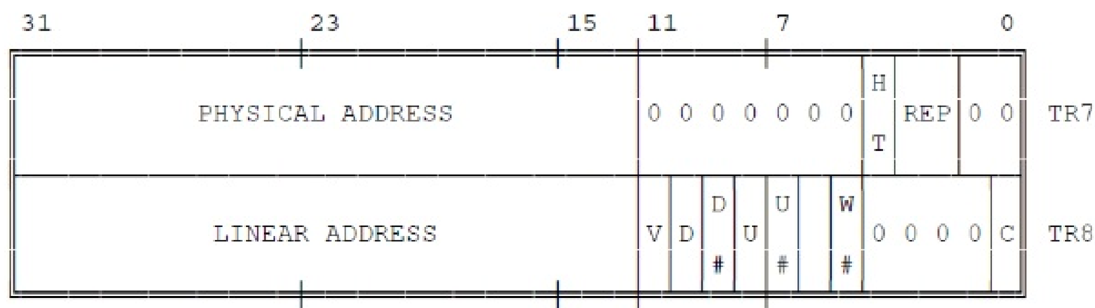


Figure 10-4. Test Registers



NOTE: 0 INDICATES INTEL RESERVED. NO NOT DEFINE

10.6.3 测试操作

写TLB项：

1，向TR7中写入双字，包含需要的物理地址，HT，和REP值。HT必须为1。REP必须指向要放置该项的通道。 2，向TR6写入双字，包含合适的线性地址，以及V, D, U和W值。确保C=0。

要当心，不要写入重复的标签；这样做的结果未定义。

检查（读）TLB项：

1，向TR6写入双字，包含适当的线性地址和属性。确保C=1。 2，保存TR7。如果设置了HT，则其他值表示读到的TLB项的内容。如果没有设置HT，则其他值无法确定。

对于测试而言，V标志的作用类似于地址标志位。通常在检查的时候置位此标志位，使没有初始化的标签不参加匹配。如果不设置，则未初始化的标签会导致不可预料的结果。

第十四章 80386实地址模式

在8086，8088，或者80188处理器下的可执行目标代码，或在80286实地址模式下的可执行目标代码均可在80386的实地址模式执行。

事实上，80386在这种模式下的体系结构与8086，8088还有80188是一致的。对程序员而言，实地址模式下的80386好像具有高速指令集和寄存器的80286。第二章和第三章介绍了这种体系结构的主要特性。

本章讨论一些附加主题，以完善系统程序员对于实地址模式下80386的整体印象。

- 地址构成。
- 寄存器和指令的扩展。
- 异常和中断的处理。
- 进入与离开实地址模式。
- 实地址模式的异常。
- 与8086的不同。
- 与80286实地址模式的不同。

14.1 物理地址构成

80386为8086程序提供了1M + 64Kbyte的存储器空间。段变换与8086中类似：段选择符向左移动4位构成段基址。有效地址的高4位补零后与段机制相加构成线性地址，见图14－1。（线性地址就是物理地址，因为没有启用页。）与80286不同的是，相加后的线性地址可以有21位有效位。在段基址和有效地址相加后有可能产生进位。在8086，进位位被截断，而在80386，进位位则被存储在线性地址的D20位。

不同于8086和80286，可以产生32位的有效地址（通过使用地址长度前缀）；然而，当地址超过65536时，会长生异常。为了和80286实地址模式完全兼容，如果有效地址超出了65536将产生伪保护错误（不带错误码的中断12和13）。

14.2 寄存器和指令

实地址模式下的寄存器集合包括8086定义的所有寄存器，加上80386新引入的寄存器：FS, GS, 调试寄存器，控制寄存器，和测试寄存器。可以显式的使用段寄存器FS和GS作为操作数，而且可以使用新引入的段一重写前缀来利用FS和GS来计算地址。指令可以利用操作数长度前缀来使用32位操作数。

保护模式下操作，检查80386选择符和描述符的指令导致未定义操作码陷阱（中断6）；这些指令包括：VERR, VERM, LAR, LSL, LTR, STR, LLDT和SLDT。在实地址模式下执行的程序可以通过80186/80188, 80286和80386的介绍来使用新加入的应用导向指令。

- New instructions introduced by 80186/80188 and 80286.
  - PUSH immediate data
  - Push all and pop all (PUSHA and POPA)
  - Multiply immediate data
  - Shift and rotate by immediate count
  - String I/O
  - ENTER and LEAVE
  - BOUND
- New instructions introduced by 80386.
  - LSS, LFS, LGS instructions
  - Long-displacement conditional jumps
  - Single-bit instructions
  - Bit scan
  - Double-shift instructions
  - Byte set on condition
  - Move with sign/zero extension
  - Generalized multiply
  - MOV to and from control registers
  - MOV to and from test registers
  - MOV to and from debug registers

14.3 中断和异常处理

80386实地址模式下的中断和异常处理与8086是一样的。中断和异常通过中断表指向处理函数。处理器将中断和异常标识符乘以4来获得其在中断表的索引。中断表是指向处理函数的长指针。当中断发生时，处理器将CS:IP压栈，关中断，清TF（单步标志位），然后将控制权交给中断表指向的函数。在处理函数末尾的IRET执行相反的过程，并将控制权交还给被中断的进程。

80386的中断处理与8086的最大不同之处在于中断表的位置和大小，这取决于IDTR（IDT寄存器）的内容。通常，这对于程序员来说没有影响，因为在上电复位后，IDTR被设置成基地址等于0，上限值等于3FFH，这与8086是兼容的。不过可以在实地址模式下使用LIDT指令来改变基地址和上限值。有关IDTR和LIDT，SIDT指令的详细信息参加第九章。如果中断表的值超出了IDTR中设置的上限值，处理器产生异常8。

14.4 进入和离开实地址模式

RESET引脚被触发后将进入实地址模式。即使系统要进入保护模式下运行，程序的在刚开始时也要临时运行在实地址模式下，这时可以为进入保护模式做一些初始化。

14.4.1 切换到保护模式

离开实地址模式的唯一方法就是切换到保护模式。当用MOV指令将CR0的PE（保护使能）置位后，处理器进入保护模式。（为了和80286兼容，也可以使用LMSW指令来设置PE位。）

关于切换到保护模式的其他描述参见第十章“初始化”。

14.5 切换回实地址模式

处理器可以通过使用MOV指令复位CR0的PE位来重新回到实地址模式。然而，如果想这么做，必须按照下面的方法：

1. 如果开启了页管理，按照下面的步骤做：

   - 变换线性地址，使它对等映射；即，线性地址等于物理地址。
   - 清除CR0的PG位。
   - 将CR3清零来清空页缓存。

2. 段变换上限值为64K（FFFFH）。实地址模式下CS的上限值需要这样设置。

3. 载入段寄存器SS, DS, ES, FS和GS。选择符指向的描述符包含下面的值，适用于实地址模式：

   - Limit = 64K (FFFFH)
   - Byte granular (G = 0)
   - Expand up (E = 0)
   - Writable (W = 1)
   - Present (P = 1)
   - Base = any value

4. 关中断。CLI关闭INTR中断。NMIs可以通过外围电路来关闭。

5. 清PE位。

6. 用长跳转JMP跳转到将要执行的实地址模式代码。该动作会刷新指令队列并给予CS寄存器合适的访问权限。

7. 使用LIDT来载入实地址模式下中断表的基地址和上限值。

8. 开中断。

9. 载入实地址模式需要的寄存器。

14.6 实地址模式异常

80386在实地址模式下报告的异常与保护模式大不相同。表14－1详细描述了实地址模式异常。

14.7 与8086的不同

通常，80386会正确的执行ROM中为8086，8088，80186和80188设计的软件。下面是一些8086和80386在执行过程中的一些细微区别。

1. 指令时钟计数。

大部分指令的执行过程，80386花费的时钟要比8086/8088要少。 这主要影响到一下几个方面：

- I/O操作要求的延时。
- 操作并口连接的80387过程中设定的延时。

2. DIV指令的除法异常触发点。

80386的除法异常总是将CS:IP指向失败的指令。8086/8088指向下一条指令。

3. 未定义的8086/8088操作码。

执行在8086/8088中未定义的操作码将导致异常6或者执行80386中定义的新指令。

4. PUSH SP写入值。

80386在执行PUSH SP时放入堆栈的值与8086/8088不同。80386把增加SP作为指令的一部分，在SP改变之前将其入栈；8086/8088在增加SP之后再将其入栈。如果入栈的值很重要，用下面的三条指令代替PUSH SP：

```
PUSH BP
MOV BP, SP
XCHG BP, [BP]
```

上面的代码在80386中的执行结果和8086/8088中的PUSH SP。

5. 超过31位的移位或循环移位。

80386将所有移位和循环移位操作计数的低位5位作为掩码。这种用32取模的操作将计数值限制为最多31位，以此来限制在执行中的代码被中断的时间。

6. 冗余前缀。

80386限制指令的最大长度为15个字节。突破这个限制的唯一方法是在指令的前面使用冗余前缀。超过长度限制的指令将导致异常13。8086/8088没有指令长度限制。

7. 操作数越过0或65536。

对于8086，试图访问的内存操作数的偏移值如果越过了65536（比如，MOV使用偏移值65536）或者0（比如，当SP＝1时使用PUSH）将导致偏移值回绕到对65536取模后的值。80386在这种情况下将产生异常－13，如果使用了数据寄存器（比如，CS, DS, ES, FS或GS）,－12，如果使用了堆栈寄存器（比如，SS）。

8．顺序执行越过了偏移值65536。

对于8086，顺序执行的指令越过了偏移值65536，处理器将取同一个段内的0地址处的下一条指令。这种情况下，80386将产生异常13。

9．某些指令禁止使用LOCK。

LOCK前缀以及相应的输出信号应该只是用来阻止总线控制器在数据移动过程中被中断。80386在使用XCHG指令和存储器交互时总是声明LOCK信号（即使没有使用LOCK前缀）。LOCK应该只是用在更新存储器的下列指令之前：BTS, BTR, BTC, XCHG, ADD, ADC, SUB, SBB, INC, DEC, AND, OR, XOR, NOT和NEG。在其他任何指令之前使用LOCK将导致未定义操作码异常（中断6）。

10．单步执行外部中断处理函数。

80386单步异常的优先级不同于8086/8088。这个改变使得在程序在单步执行时阻止外部中断进入后被单步执行。80386单步异常的优先级高于任何外部中断。由INT指令或异常产生的中断进入处理函数后，80386仍将单步执行。

11．IDIV指令80H或8000H商异常。

80386的IDIV指令的商可以是最大的负数。8086/8088产生异常0。

12．堆栈中的标志位。

由PUSHF，中断，异常存储的标志位在位12－15上与8086不同。在8086上，它们被作为一个整体，而在80386上，位15总是0，位14－12反映的是最后装入它们的值。

13．NMI中断和NMI处理函数。

在80386识别出NMI中断后，NMI中断保持屏蔽，直到执行了IRET指令。

14．协处理器错误指向中断16。

任何带有协处理器的80386系统必须使用中断向量16来指向协处理器错误异常。如果8086/8088使用另一个向量，那么这两个向量都要指向协处理器错误异常处理函数。

15．数字异常处理函数应该允许前缀。

在80386上，为协处理器异常保存的CS:IP指向ESC指令之前的任何前缀。在8086/8088上，保存的CS:IP指向ESC指令。

16．协处理器不使用中断控制器。

协处理器错误信号在80386上不会进入中断控制器（8087 INT信号这样做）。如果在协处理器错误处理函数中有些指令是用来处理中断控制器的，就需要删除它们。

17．6个新的中断向量。

80386增加了6个异常，它们只有在8086程序有隐含错误的时候才发生。建议按照非法操作来增加这些异常的处理函数。这些增加的代码不会明显的影响现有的8086软件，因为这些中断正常情况下不会发生。这些中断标识符应该还没有被8086软件使用，因为它们被Intel放在了保留区域。表14－2描述了这6个新异常。

18．1M地址回绕。

实地址模式下，80386在1M地址外不会回绕。在8086家族中，可能声明的地址超过1M大小。例如，选择符是0FFFFH，偏移值是0FFFFH，有效地址将是10FFFFH（1M + 65519）。8086，最长20位地址，截断高位。然而80386可以有32位长地址，因此不会截断这样的地址。

Table 14-1. 80386 Real-Address Mode Exceptions

```
Description Interrupt Function that Can
Return Address
Number Generate the Exception
Points to Faulting
Instruction
Divide error 0 DIV, IDIV
YES
Debug exceptions 1 All
Some debug exceptions point to the faulting instruction, others to the
next instruction. The exception handler can determine which has occurred by
examining DR6.
Breakpoint 3 INT
NO
Overflow 4 INTO
NO
Bounds check 5 BOUND
YES
Invalid opcode 6 Any undefined opcode or LOCK
YES
used with wrong instruction
Coprocessor not available 7 ESC or WAIT
YES
Interrupt table limit too small 8 INT vector is not within IDTR
YES
limit
Reserved 9-12
Stack fault 12 Memory operand crosses offset
YES
0 or 0FFFFH
Pseudo-protection exception 13 Memory operand crosses offset
YES
0FFFFH or attempt to execute
past offset 0FFFFH or
instruction longer than 15
bytes
Reserved 14,15
Coprocessor error 16 ESC or WAIT
YES
Coprocessor errors are reported on the first ESC or WAIT instruction
after the ESC instruction that caused the error.

Two-byte SW interrupt 0-255 INT n
NO
```

Table 14-2. New 80386 Exceptions

Interrupt	Function
Identifier	
5	BOUND指令以一个超出上限的寄存器值被执行。
6	未定义的操作码或LOCK应用与错误的指令。
7	执行ESC指令时，MSW中EM被置位。执行WAIT指令时TS被置位。
8	异常或中断向量超出了IDTR中定义的上限值；只有在用LIDT指令修改上限值才有可能发生这个错误，因为默认的3FFH对于256个中断ID足够了。
12	操作数越过了堆栈段边界，比如，用偏移值0FFFFH执行MOV或者SP=1时，执行压栈指令PUSH, CALL或INT。
13	操作数越过了段边界，不包括堆栈段；或者顺序指令执行试图跨越偏移值0FFFFH；或者指令长度超过了15字节（包括前缀）。

14.8 与80286实地址模式的不同

80386的实地址模式和80286几乎没有什么不同，除了初始化过程外，对已有的80286程序不太可能有影响。

14.8.1 总线锁

80286和80386有着不同的总线锁实现方法。使用专属与80286的存储器锁的程序如果被传到80386的某个应用上，可能不会正常运行。

LOCK前缀以及相应的输出信号应该只是用来阻止总线控制器在数据移动过程中被中断。LOCK应该只是用在更新存储器的下列指令之前。在其他任何指令之前使用LOCK将导致未定义操作码异常。

- 位测试与修改：BTS, BTR, BTC。
- 交换：XCHG。
- 一元算术和逻辑：INC, DEC, NOT和NEG。
- 二元算术和逻辑：ADD, ADC, SUB, SBB, AND, OR, XOR。

锁指令只被授权由目的操作数定义的存储器区域，但也可以锁定一个更大的存储器区域。例如，典型的8086和80286配置锁定整个物理存储器空间。对于80386，定义的存储器区域被授权锁，以防止处理器在完全相同的区域执行锁指令，即相同的起始地址和相同的长度。

14.8.2 第一条指令的位置

80386的起始位置是0FFFFFFF0H（距32位地址空间末端16字节），不同于80286的0FFFFF0（距24位地址空间末端16字节）。许多80286 ROM初始化程序可以在这个环境下正常运行。其他的可以通过重定义外部硬件的A{31-20}来正常工作。

14.8.3 通用寄存器的初始值

80386的某些通用寄存器在复位后可能含有与80286不同的值。这不应该回引起兼容问题，因为8086的寄存器在复位后是未定义的。如果在上电过程中要求自检，并且检测到了错误，则EAX包含一个非零值。EDX包含部件和版本标识符。更多信息参见第十章。

14.8.4 MSW初始化

80286将MSW初始化成FFF0H，但是80386用0000H来初始化这个寄存器。这个差异应该没关系，因为不同的比特位在80286中未定义。读取MSW的程序只有在它们依赖于那些高位的未定义位的时候，才会在80386上有不同的表现。